



Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR
DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL:

**REFACTORING PLANNING FOR DESIGN SMELL
CORRECTION IN OBJECT-ORIENTED SOFTWARE**

Presentada por Francisco Javier Pérez García para optar al grado
de
doctor por la Universidad de Valladolid

Dirigida por:
Dra. Yania Crespo González-Carvajal

a mi madre

*“The beauty of a living thing is not the atoms that go into it,
but the way those atoms are put together.” **Carl Sagan***

Resumen

La evolución de un sistema software suele conllevar cierto grado de deterioro de su estructura. Esto ocurre principalmente porque el mantenimiento se centra más en la corrección de *bugs* y en la inclusión de nuevas funcionalidades que en el seguimiento y la mejora de su arquitectura y diseño. Las malas prácticas de diseño, causadas a menudo por la inexperiencia, el conocimiento insuficiente o la urgencia, originan *design smells* (“malos olores en el diseño”): problemas en la estructura de un sistema, que no producen errores de compilación o de ejecución, pero que afectan negativamente a los factores de calidad del software.

Las refactorizaciones constituyen una técnica clave en la evolución del software. Pueden emplearse para mejorar la estructura y la calidad de un sistema sin modificar su comportamiento observable. De ahí que parecen ser también la técnica más apropiada para corregir *design smells*. La mayoría de enfoques de gestión de *design smells* se centran en sugerir cuáles serían los cambios y las estructuras más adecuadas para remediar un mal olor y recuperar la intención del diseño original. Estas sugerencias se suelen dar en términos de refactorizaciones.

Una de las dificultades que presenta la aplicación de estas sugerencias de refactorización radica en que es inusual que las precondiciones de las refactorizaciones se cumplan en el momento deseado de su aplicación. El desarrollador, por lo tanto, tiene que planificarlas con antelación y aplicar una cantidad significativa de cambios adicionales para resolver esto. En un proceso de refactorización que persiga un objetivo complejo, como la corrección de *design smells*, es prácticamente inevitable la aparición de este problema.

La presente Tesis Doctoral está orientada a mejorar la automatización de las actividades de refactorización enfocadas a la corrección de *design smells*. La propuesta desarrollada se basa en la definición de estrategias de refactorización y en la instanciación de planes de refactorización a partir de estas. Las estrategias de refactorización son especificaciones automatizables de secuencias de refactorización complejas dirigidas a un objetivo particular, como la corrección de *design smells*. Los planes de refactorización son secuencias de refactorización instanciadas a partir de estrategias de refactorización, que pueden ser aplicadas sobre un sistema para conseguir de forma efectiva ese objetivo. Las estrategias y los planes de refactorización permiten computar las refactorizaciones preparatorias requeridas, ayudando al desarrollador a resolver el incumplimiento de las precondiciones.

La planificación automática es una rama de la inteligencia artificial orientada a calcular las secuencias de acciones que se deben realizar para lograr un objetivo determinado. El estudio que se presenta demuestra que es una técnica apropiada para instanciar planes de refactorización y, por lo tanto, para respaldar procesos de refactorización complejos. Hemos observado que la planificación mediante redes de tareas jerárquicas proporciona, de entre los diferentes enfoques de planificación existentes, el mejor equilibrio entre estrategias procedimentales y basadas en búsqueda para el problema de planificación de refactorizaciones. Así, la investigación que se presenta en este documento, muestra que la generación de planes de refactorización –basados en planificación de redes de tareas jerárquicas– es un enfoque muy apropiado para ayudar a la automatización de la actividad de corrección de *design smells*.

Palabras clave: refactorización, *bad smells*, *design smells*, corrección de *design smells*, estrategias de refactorización, planes de refactorización, planificación automática, planificación de redes de tareas jerárquicas.

Abstract

The evolution of a software system often implies a certain degree of deterioration of the system's structure. This mainly happens because maintenance efforts concentrate more on bug correction and the addition of new functionality, than on the control and improvement of the system's architecture and design. Bad design practices, often due to inexperience, insufficient knowledge or time pressure, are at the origin of design smells. Design smells are problems encountered in the system's structure which do not produce compile or run-time errors, but negatively affect software quality factors. Correcting or, at least, reducing design smells can improve software quality.

Refactoring is a key technique in software evolution. It can be used to improve the structure and quality of a software system without changing its observable behaviour. Consequently, refactoring also seems to be the most adequate technique to correct design smells. Most design smell management approaches focus on suggesting which are the best redesign changes to perform, and which are the best structures to remedy a smell in order to recover the original design intent. These suggestions are often given in terms of refactorings.

One of the difficulties in applying refactoring operations is that it is rare that the preconditions of the desired refactorings could be fulfilled by the system's source code in its current state. Therefore, the developer has to plan ahead and apply a significant amount of additional changes to solve this problem. This is a recurring issue appearing when a refactoring process pursues a complex objective, such as correcting design smells.

This PhD Thesis Dissertation is aimed at improving the automation of the refactoring activity when it is oriented to the correction of design smells. The approach is based on the definition of refactoring strategies and the instantiation of refactoring plans from them. Refactoring strategies are specifications of complex refactoring sequences aimed at a certain goal, such as the correction of design smells, that can be easily automated. Refactoring plans are refactoring sequences, instantiated from refactoring strategies, that can be applied to the system to effectively achieve that goal. Refactoring strategies and refactoring plans allow the necessary preparatory refactorings to be computed, thus helping the developer to circumvent the violation of refactoring preconditions.

Automated planning is an artificial intelligence branch that seeks to generate sequences of actions that will achieve a certain goal when they are performed. The study presented in this document demonstrates that automated planning is a suitable technique for instantiating refactoring plans and thus, for supporting the problem of performing complex refactoring processes. We have found that hierarchical task network planning provides, among all the existing planning approaches, the best balance between search-based and procedural-based strategies for the problem of refactoring planning. Therefore, this research introduces the generation of refactoring plans –based on hierarchical task network planning– as an approach to support the automation of the design smells correction activity.

Keywords: refactoring, bad smells, code smells, design smells, design smell correction, refactoring strategies, refactoring plans, automated planning, hierarchical task network planning.

Acknowledgments

First of all, I would like to thank my advisor Yania Crespo, for her endless patience, support and above all, for being a true friend.

I would also like to thank all my colleagues and friends at the GIRO research group, and at the Department of Computer Science, of the University of Valladolid, especially Esperanza Manso, Carlos López, Raúl Marticorena Miguel Ángel Laguna, Félix Prieto, José Manuel Marqués, Diego Llanos, Arturo González, Margarita Gonzalo, Miguel Ángel Villaroel and Carlos Alonso, and all the rest of you who have helped me in this way. I hope each of you know why I should be grateful to you for. In case you do not know, just ask.

I would also like to thank Tom Mens, Naouel Moha, Günter Kniessel and Juan Ramil, whose help has been essential in this process. I hope we'll keep working together from now on.

I would like to thank the students who have carried out programming projects related to this Thesis. I hope you are doing well despite of me torturing you.

And, of course, I want to thank my family and friends. Hope I can now spend more time with you.

Finally, I want to thank María, because walking this path together has made everything easier.

Francisco Javier Pérez García,
Valladolid, 6 de mayo de 2011

This work has been partially funded by the regional government of *Castilla y León* (project VA-018A07) and by the Spanish government (*Ministerio de Ciencia e Innovación*, project TIN2008-05675).

Typographical Notes

The following typographical conventions are used in this dissertation:

- Names of programming languages, tools, products, etc. are written in smallcaps font, *eg.* LISP, JAVA, ECLIPSE, etc.
- Names of refactorings are capitalized and written in bold style, *eg.* **MOVE METHOD**.
- Names of design smells are written in italics, *eg.* *Large Class*.
- Code listings and excerpts are written in typewriter font, *eg.* `(create-getter ?package ?class ?field ?getter-name)`.
- Elements of a model are written in bold italics, *eg.* ***Precondition***, ***Entity***.
- Quotes from other authors are written in italics, enclosed between double quotes, *eg.* “*this is a quote*”, and placed in a separate paragraph when significantly relevant or large, *eg.*

“This is a large quote”

PART I

PLANIFICACIÓN DE REFACTORIZACIONES PARA LA CORRECCIÓN DE DESIGN SMELLS EN SOFTWARE ORIENTADO A OBJETOS

(RESUMEN EN CASTELLANO)

Índice general

1. Introducción	1
1.1. Visión general del problema	3
1.1.1. ¿Cuál es el problema que se quiere resolver?	3
1.1.2. ¿Por qué es un problema?	3
1.1.3. ¿Por qué es un problema importante?	4
1.2. Tesis, objetivos y contribuciones	7
1.2.1. Objetivos	7
1.2.2. Resumen de contribuciones	7
1.3. Estructura de la tesis	8
2. Contexto	11
2.1. Design smells: definiciones y terminología	11
2.1.1. Sobre el término “ <i>smell</i> ”	12
2.1.2. Code smells	12
2.1.3. Design smells	13
2.1.4. Revisión histórica de los design smells	15
2.2. Refactorizaciones	17
2.2.1. Aplicación de refactorizaciones	18
2.2.2. Desarrollo de herramientas de refactorización	19
2.2.3. Minería de refactorizaciones	19
2.2.4. Sugerencias de refactorización	20
2.3. Corrección de design smells mediante refactorizaciones	21
3. Estudio exhaustivo de la gestión de design smells	23
3.1. Visión general del estudio	23
3.1.1. Modelos de características	24
3.1.2. Características principales de la taxonomía	25
3.2. Design smell	26
3.3. Artefacto	28
3.3.1. Tipo de artefacto	28
3.3.2. Versiones	29
3.3.3. Tipo de representación	29
3.4. Actividad	30
3.4.1. Especificación	31
3.4.2. Detección	32
3.4.3. Visualización	34

3.4.4.	Corrección	34
3.4.5.	Análisis del impacto	35
3.5.	Conclusiones del estudio	37
4.	Estrategias de Refactorización	39
4.1.	Análisis de especificaciones de corrección de design smells	39
4.1.1.	Especificaciones de design smells en la actualidad	39
4.1.2.	Un modelo para las especificaciones de corrección actuales	47
4.1.3.	Especificaciones de refactorizaciones en la actualidad	50
4.1.4.	Actividades de aplicar una estrategia de corrección de design smells	52
4.2.	Problemas sin resolver en la automatización de estrategias de corrección	53
4.2.1.	Aplicabilidad de las refactorizaciones	53
4.2.2.	Descripciones no formales de estrategias de corrección	54
4.3.	Definición de estrategias de refactorización	55
4.3.1.	Visión general de las estrategias de refactorización	55
4.3.2.	Un modelo de refactorizaciones y otras transformaciones	57
4.3.3.	Un modelo de estrategias de refactorización	59
4.4.	Un lenguaje para especificar estrategias de refactorización	63
4.5.	Características del problema	69
5.	Planificación de Refactorizaciones	73
5.1.	Trabajo previo	73
5.2.	Breve introducción a la planificación automática	77
5.2.1.	Conceptos básicos de planificación automática	79
5.2.2.	El modelo restringido de la planificación clásica	81
5.2.3.	Caracterización del problema de planificación de refactorizaciones	82
5.2.4.	Tipos de planificadores	85
5.3.	JSHOP2: un planificador de redes jerárquicas	89
5.3.1.	Planificación HTN	89
5.3.2.	Características de JSHOP2	91
5.3.3.	Elementos de un problema de planificación en JSHOP2	92
5.3.4.	Gestión de variables	99
5.4.	Estrategias de refactoring como HTNs	102
5.4.1.	Elementos del sistema	103
5.4.2.	Consultas del sistema	103
5.4.3.	Pasos de transformaciones NBPT	104
5.4.4.	NBPT	112
5.4.5.	Refactorizaciones	113
5.4.6.	Invocaciones de consultas y estrategias de refactorización	113
5.4.7.	Alternativas no-deterministas	114
5.4.8.	Bucles no-deterministas	115
5.4.9.	Pasos de estrategias sin ordenar	117
5.4.10.	Lenguaje de especificación de estrategias como HTNs de JSHOP2	119
5.5.	Computando un problema de planificación de refactorizaciones	119
5.6.	Cómo JSHOP2 permite planificar refactorizaciones	121
5.6.1.	Requisitos generales	121
5.6.2.	Requisitos como problema de planificación	122

6. Estudio de casos	125
6.1. Un dominio de planificación de refactorizaciones	125
6.1.1. Refactorizaciones	125
6.1.2. Estrategias de refactorización	131
6.2. Prototipo de planificador de refactorizaciones	146
6.2.1. Herramientas utilizadas en el prototipo	146
6.2.2. Integración de las herramientas en nuestro prototipo	149
6.3. Descripción del estudio de casos	151
6.3.1. Configuración de los experimentos	151
6.4. Análisis de los resultados	155
6.4.1. Discusión de los planes producidos	155
6.4.2. Discusión de la eficiencia de los prototipos	159
6.5. Conclusiones del estudio de casos	172
6.6. Caracterización del enfoque de planificación de refactorizaciones	174
6.6.1. Según los design smells	174
6.6.2. Según los artefactos	175
6.6.3. Según las actividades	175
7. Conclusiones	177
7.1. Resultados generales	177
7.2. Resultados relacionados con el enunciado de la tesis	178
7.3. Resultados relacionados con los objetivos de la tesis	180
7.4. Caracterización de la propuesta desarrollada	181
7.5. Comparación con otros trabajos relacionados	181
7.6. Limitaciones de la propuesta desarrollada	183
7.7. Resumen de contribuciones	184
7.8. Trabajo futuro y preguntas abiertas	185
Referencias	186
A. Especificación del lenguaje de planificación JSHOP2	205
A.1. Símbolos	205
A.2. Términos	205
A.3. Átomos y expresiones lógicas	206
A.4. Precondiciones	207
A.5. Axiomas	207
A.6. Átomos de tareas y listas de tareas	207
A.7. Operadores	208
A.8. Métodos	208
A.9. Dominios de planificación	208
A.10. Problemas de planificación	209
B. Traducción de los lenguajes de estrategias de planificación	211
B.1. Definiciones de estrategias	211
B.2. Secuencias de pasos	212
B.3. Gestión de variables entre métodos	214
B.4. Gestión de listas	215

B.5. Consultas, invocaciones y expresiones booleanas	215
B.6. Variables y literales	216
B.7. Alternativas	217
B.8. Bucles: mientras	218
B.9. Bucles: para	219
B.10. Bucles: para	221
B.11. Pasos sin ordenar	223
C. El dominio de planificación de refactorizaciones	225
C.1. Notación	225
C.2. Estrategias de refactorización	225
C.3. Refactorizaciones	226
C.4. Operadores	226
C.5. Procedimientos Java externos	227
C.6. Consultas del sistema	227

Capítulo 1

Introducción

La evolución del software es una parte fundamental del proceso de desarrollo que deriva con frecuencia en un aumento de la entropía del software y, en consecuencia en el deterioro de su estructura [EGK⁺01]. Esto ocurre principalmente porque los esfuerzos de mantenimiento se centran más en la corrección de *bugs* y en la incorporación de nuevas funcionalidades que en la vigilancia y la mejora de la arquitectura y el diseño del sistema [Bro75].

Los problemas en la estructura del software pueden manifestarse como *design smells* (“malos olores en el diseño”) [BF99], que son problemas de diseño provocados por malas decisiones de diseño, que conducen a un software con una estructura deficiente. Esto puede perjudicar posteriormente el desarrollo y el mantenimiento¹ del software, ya que provoca que resulte más complejo para los desarrolladores introducir cambios en el mismo. Los *design smells* no producen errores en tiempo de compilación o ejecución, pero afectan negativamente a los factores de calidad del software. De hecho, en el futuro pueden llegar a provocar realmente errores en tiempo de compilación y ejecución. Corregir, o al menos, reducir los *design smells* puede mejorar la calidad del software. Lo más deseable es prevenir estos problemas, pero aún así, sigue siendo necesario disponer de técnicas para la detección y corrección de *design smells* en el caso de que se produzcan. Dado que el objetivo de la corrección de *design smells* no es modificar el comportamiento del sistema —el sistema funciona correctamente—, los *design smells* se corrigen mediante refactorizaciones.

Las refactorizaciones son transformaciones del código fuente que modifican el diseño de un sistema al tiempo que preservan su comportamiento observable [Opd92]. Pueden emplearse de forma general para mejorar ciertos factores de calidad como reusabilidad, comprensibilidad, mantenibilidad, etc. Como un objetivo más concreto, las refactorizaciones pueden ayudar a dotar al sistema de una determinada estructura, por ejemplo, aplicar un patrón de diseño [Ker04] o para consolidar la arquitectura del sistema [NL06].

Las operaciones de refactorización se definen en términos de precondiciones y transformaciones. Las precondiciones definen las situaciones en las que una operación de refactorización puede ser ejecutada con la seguridad de no alterar el comportamiento del sistema. Las especificaciones de transformaciones describen los cambios que se han de aplicar al sistema. Las operaciones de refactorización están pensadas para ser ejecutadas en pequeños pasos, de forma que las refactorizaciones más complejas pueden estar compuestas de otras más simples. La preservación del comportamiento también es más sencilla de verificar en las refactorizaciones más sencillas.

¹La mantenibilidad del software es la facilidad con la que un sistema o componente software puede ser modificado para corregir defectos, mejorar su rendimiento u otras características, o para adaptarlo a posibles modificaciones del entorno [ISO01].

Cuando un proceso de refactorización se orienta a resolver un problema complejo, como la corrección de *design smells*, se requiere aplicar un número significativo de cambios. Las precondiciones de una refactorización ayudan a garantizar la preservación del comportamiento, pero al mismo tiempo obstaculizan la aplicación de una refactorización compleja al restringir la aplicabilidad de las operaciones que componen la transformación. Si alguna de las precondiciones, de alguna de las operaciones incluidas en la secuencia de transformación prevista, no se cumple en el momento de su aplicación, entonces no se puede aplicar la secuencia de transformación en su conjunto. Esto hace que para el desarrollador sea más complicado realizar procesos de refactorización complejos, como la corrección de *design smells*.

Algunos autores han intentado resolver este problema [KK04]. La composición estática de refactorizaciones permite evitar la violación de las precondiciones de las refactorizaciones intermedias cuando estas forman parte de secuencias predefinidas. En el caso de la corrección de *design smells*, la definición de las refactorizaciones que se deben aplicar se realiza en términos de estrategias de corrección, que son principalmente heurísticas. La aplicación de este tipo de secuencias de refactorizaciones tiene una dificultad añadida, ya que la estrategia de corrección prevista debe ser instanciada para cada situación particular. Esto significa que el desarrollador tiene que encontrar una secuencia de refactorizaciones viable para cada caso, de entre las múltiples opciones definidas por la estrategia de corrección.

Se han desarrollado muchas técnicas para asistir las diferentes actividades de la gestión de *design smells*. Por ejemplo, las métricas y análisis estructurales han demostrado ser muy útiles para la detección de *design smells* y para proponer los posibles cambios que se pueden aplicar para reducir o eliminar estos problemas. Una manera adecuada de realizar estas sugerencias es mediante propuestas de refactorizaciones. El problema, como ya se ha mencionado, reside en qué es poco frecuente que el código fuente del sistema cumpla las precondiciones de todas las refactorizaciones sugeridas. Esto implica que suele ser necesario aplicar ciertas transformaciones adicionales, con el fin de “preparar” el sistema para poder realizar los cambios deseados. Por lo tanto, las sugerencias de refactorizaciones no son suficientes para permitir la corrección sistemática de *design smells*. Para este objetivo, la revisión del estado del arte en cuanto a la corrección de *design smells* revela que existe la necesidad de una propuesta que permita planificar y obtener estrategias de corrección de *design smells* ejecutables. Las sugerencias de refactorización que producen los enfoques existentes, tal y como son generadas, no se pueden ejecutar.

La presente tesis doctoral aborda este problema. Esta investigación está enfocada a mejorar la automatización de las actividades de refactorización cuando están orientadas a la corrección de *design smells*. De hecho, cualquier actividad en la que intervengan procesos de refactorización complejos puede ser mejorada si podemos proporcionar un método, más automático que los existentes, para calcular las secuencias de refactorizaciones requeridas para conseguir un determinado objetivo. El enfoque propuesto se basa en la generación de planes de refactorización. Hemos definido los planes de refactorización como especificaciones de secuencias de refactorización que se corresponden con una propuesta de rediseño o una sugerencia de corrección de un sistema, y que pueden ser generadas para aplicar una estrategia de corrección de *design smells* en cada caso particular. De este modo, las refactorizaciones deseadas pueden ser directamente aplicadas sobre el código fuente del sistema. Los planes de refactorización permiten calcular las refactorizaciones preparatorias necesarias, ayudando al desarrollador a evitar el problema del incumplimiento de las precondiciones de las refactorizaciones deseadas.

Teorías formales, como la transformación de grafos o la lógica de primer orden podrían ser utilizadas para analizar un conjunto de refactorizaciones, dentro de un contexto dado por

el código fuente de un sistema y una propuesta de rediseño, con el fin de generar planes de refactorización. Nosotros mismos hemos explorado un enfoque basado en transformación de grafos antes de desarrollar la propuesta que se presenta en esta tesis. Esta exploración previa nos ha servido para profundizar en nuestra comprensión y conocimiento del problema. Esta tesis doctoral analiza el estado del arte de la corrección de *design smells*, lo que nos sirve para definir las características principales del problema y para modelar una solución mediante la introducción del concepto de **planes de refactorización**. El enfoque que se presenta en esta tesis, para mejorar la automatización de las tareas de refactorización, se basa en planificación automática [GNT04], y, más concretamente, en planificación de redes jerárquicas de tareas (*hierarchical task network planning* – HTN) [GNT04, capítulo 11]. Esta técnica es analizada para demostrar su viabilidad y, finalmente, mostramos como puede ser empleada para la generación de planes de refactorización y demostramos su aplicación mediante un estudio de casos.

1.1. Descripción general del problema

1.1.1. ¿Cuál es el problema que se quiere resolver?

El problema que se quiere resolver en esta tesis es el de proporcionar un soporte automático o semi-automático para la planificación de refactorizaciones con el fin de corregir *design smells* en software orientado a objetos. Este problema implicará la definición de un marco para la instanciación de sugerencias de eliminación de *design smells* en planes de corrección que puedan ser aplicados de manera efectiva mediante refactorizaciones, ya que se debe preservar el comportamiento del sistema. El problema puede enmarcarse en un ámbito más amplio. Los resultados derivados de esta tesis pueden ser de utilidad en otros escenarios en los que se pretenda aplicar secuencias complejas de refactorización dirigidas a lograr un determinado objetivo o mejora de diseño.

El principal problema que se pretende resolver es el de proporcionar un soporte automático o semi-automático para la planificación de refactorizaciones en el contexto de la corrección de design smells.

1.1.2. ¿Por qué es un problema?

Las técnicas de detección y corrección de *design smells* están madurando y aumentando en número. Por citar algunas, podemos encontrar detección de *bad smells* mediante métricas [LM06, CLMM06], uso de patrones estructurales para identificar defectos de diseño [Moh08] o análisis de conceptos formales/relacionales para sugerir la reorganización de entidades en orientación al objeto [PCML03, MBG06]. El problema vigente, es que todas las sugerencias de cambios proporcionadas por los enfoques y herramientas existentes no son aplicables directamente sobre un sistema. Estas sugerencias se dan en términos de refactorizaciones que no son inmediatamente aplicables. Se necesita planificar con antelación refactorizaciones preparatorias adicionales y secuencias complejas de refactorizaciones. Independientemente del grado de detalle que presenten estas sugerencias [TSG04], sólo pueden describirse en términos generales y sigue existiendo la necesidad de instanciarlas para cada caso particular, para cada estado de un sistema. Estas descripciones de corrección de *design smells* no pueden anticipar todas las situaciones de instanciación posibles. Por lo tanto, se le deja al desarrollador la responsabilidad de instanciar estas especificaciones de corrección. Las refactorizaciones que se requiere ejecutar son a menudo

secuencias muy complejas, muy diferentes de las sugeridas originalmente por la herramienta de gestión de *design smells*.

Las operaciones de refactorización no solo necesitan ser planificadas con antelación solamente en el caso de la corrección de *design smells*, sino también cuando se utilizan de forma cotidiana para cualquier otro objetivo. Según un estudio realizado por Murphy-Hill and Black en [MHB08], es necesario mejorar la usabilidad de las herramientas de refactorización con el fin de que estas técnicas sean utilizadas por un mayor número de desarrolladores. En concreto, los autores del estudio han observado que el incumplimiento de las precondiciones es un problema que no ha sido abordado todavía. Cuando un desarrollador trata de aplicar sin éxito una refactorización con una herramienta, la mayor parte de los errores que se producen son violaciones de precondiciones. Otros problemas encontrados incluyen cómo comprender y evitar estos problemas y cómo decidir qué refactorizaciones se deben aplicar. En el estado actual de las herramientas de refactorización, la ayuda ofrecida por los entornos de desarrollo no es suficiente. Las herramientas actuales sólo muestran mensajes de error con muy pocos detalles útiles. Los autores proponen mejorar las herramientas de refactorización con mejores interfaces que puedan ayudar al desarrollador a entender los errores de incumplimiento de precondiciones.

Por lo tanto, existe una necesidad clara de mejorar la asistencia automatizada al desarrollador respecto al incumplimiento de precondiciones y respecto a cómo resolver este problema con el fin de poder aplicar las refactorizaciones deseadas.

1.1.3. ¿Por qué es un problema importante?

Refactorizar implica “aplicar muchas refactorizaciones”

El desarrollo de una solución para el problema de incumplimiento de las precondiciones de una refactorización abre las puertas a una posible mejora en la automatización de procesos de refactorización complejos, como la corrección de *design smells* mediante refactorizaciones. La generación automática de secuencias de refactorización complejas, dirigidas a habilitar el cumplimiento de las precondiciones de una refactorización, pueden ayudar en la aplicación de una única refactorización o un conjunto de ellas. Esto puede ser extendido y generalizado para planificar secuencias de refactorización orientadas, por ejemplo, a la corrección de *design smells*.

Abordar el problema del cumplimiento de las precondiciones puede ayudar a mejorar los procesos de refactorización en muchos casos. Las operaciones de refactorización no se ejecutan de forma aislada, sino que se aplican en secuencias que comprenden amplios cambios sobre el código de un sistema. Algunos ejemplos de situaciones en las que se emplean secuencias complejas de refactorización son:

- Composición de refactorizaciones, definidas por Opdyke mediante “refactorizaciones de alto nivel” (*thigh-level refactorings*) [Opd92, página 79], y “refactorizaciones compuestas” (*composite refactorings*) [Opd92, página 74]. Las refactorizaciones complejas como estas pueden construirse a partir de otras más simples. Por ejemplo, para aplicar la refactorización de alto nivel **CREATING AN ABSTRACT SUPERCLASS** [Opd92, página 79], Opdyke elabora una especificación que incluye otras de bajo nivel como: **CREATE EMPTY CLASS** [Opd92, página 56], **CREATE MEMBER FUNCTION** [Opd92, página 57], **DELETE MEMBER FUNCTIONS** [Opd92, página 59], **CHANGE VARIABLE NAME** [Opd92, página 60], **CHANGE MEMBER FUNCTION NAME** [Opd92, página 61], **CHANGE TYPE** [Opd92, página 62], **CHANGE ACCESS CONTROL MODE** [Opd92, página 63], **REORDER FUNCTION ARGUMENTS** [Opd92, página 66], **ADD FUNCTION BODY** [Opd92, página 67], **MOVE MEMBER**

VARIABLE TO SUPERCLASS [Opd92, página 72], y refactorizaciones compuestas como: **CONVERT CODE SEGMENT TO FUNCTION** [Opd92, página 75], **MOVE CLASS** [Opd92, página 76]. Las refactorizaciones compuestas, las define como, refactorizaciones que se construyen sobre otras de bajo nivel pero que son más sencillas que las de alto nivel.

- Las especificaciones de refactorizaciones de Fowler *et al.* [FBB⁺99]. Una refactorización puede habilitar o deshabilitar las precondiciones de otras. Fowler *et al.* describen en sus especificaciones las posibles dependencias entre refactorizaciones. Para poder aplicar una refactorización determinada, se deben ejecutar otras refactorizaciones que habilitan las precondiciones de la primera. Por ejemplo, la refactorización **REPLACE CONDITIONAL WITH POLYMORPHISM** [FBB⁺99, página 255] suele requerir que se ejecuten además otras como: **REPLACE TYPE CODE WITH SUBCLASSES** [FBB⁺99, página 223], **EXTRACT METHOD** [FBB⁺99, página 110], **MOVE METHOD** [FBB⁺99, página 142], etc.
- Refactorizaciones dirigidas a un objetivo determinado, como la inclusión o la eliminación de patrones de diseño [Ker04], la corrección de *bad smells* [BF99]. Como ejemplo, según Fowler *et al.*, para poder eliminar una *Data Class*, se deben aplicar una serie de refactorizaciones que comprenden: **ENCAPSULATE FIELD** [FBB⁺99, página 206], **ENCAPSULATE COLLECTION** [FBB⁺99, página 208], **REMOVE SETTING METHOD** [FBB⁺99, página 300], **MOVE METHOD** [FBB⁺99, página 142], **EXTRACT METHOD** [FBB⁺99, página 110] y **HIDE METHOD** [FBB⁺99, página 303].

Todas estas situaciones pueden beneficiarse de una técnica que permita la generación automática de planes o secuencias de refactorización, para poder construir refactorizaciones complejas. La presente tesis doctoral está motivada por el último caso enumerado, pero de forma indirecta, también podrá ser aplicada a los otros dos casos.

En lo relativo a los *design smells*

Como estableció Lehman en sus *Leyes de la evolución del software* [Leh96], es un hecho conocido y verificado que el software evoluciona, y si no lo hace se vuelve obsoleto y deja de ser útil paulativamente:

“Un programa de tipo-E² que se encuentra en uso, debe adaptarse continuamente, en caso contrario, el programa se hace progresivamente menos satisfactorio.” **I - Ley del cambio continuo**

Como se ha mencionado al comienzo de la introducción, el deterioro del diseño del software es un proceso recurrente [Bro75, EGK⁺01]. Una forma de abordar este deterioro es mediante la detección y corrección de *design smells*. Esta manera de abordar el problema es a lo que Kerievsky llama “*pagar la deuda del diseño*” [Ker04], o a lo que Neil *et al.* denominan “*refactorización estratégica*” [NL06].

²Tipo-E se refiere al software que modela procesos, organizaciones y actividades humanas. Este tipo de software evoluciona de forma natural según lo haga su dominio del problema. La “E” significa evolutivo. Otros tipos de software definidos por Lehman son: software-S, que puede ser derivado de manera formal a partir de la especificación; y software-P, que puede ser especificado y derivado de manera formal, pero la computación se efectúa mediante heurísticas y aproximación [Leh80].

Los *design smells* pueden con certeza afectar a los factores de calidad del software de forma negativa. A pesar de que, hasta dónde sabemos, no se han realizado experimentos concluyentes sobre este tema, el conocimiento común en ingeniería del software parece confirmar esta idea.

En su tesis doctoral [Mar02], Marinescu emplea modelos de calidad para establecer relaciones entre *design smells* y factores de calidad del software. A estos modelos los denomina *factor strategy models* [Mar02, páginas 87–108]. Esta correspondencia entre *design smells* y factores de calidad le permiten describir qué *design smells* pueden tener un impacto negativo sobre un determinado factor, y viceversa, qué factores pueden deteriorarse debido a un *design smell* en particular. Las relaciones definidas mediante estos modelos están ponderadas, de forma que también pueda tenerse en cuenta el grado de impacto de cada *design smells*. Este enfoque se aplica para la construcción de un modelo de mantenibilidad. Mediante este modelo, se trata de describir cómo la mantenibilidad del sistema se puede ver negativamente afectada por algunos *design smells* como: *Feature Envy*, *Temporary Field*, *Shotgun Surgery*, *Refused Bequest*, *God Class*, *God Package*, *God Method*, *Data Class*, etc.

Marinescu presenta también un estudio de casos con el fin de evaluar los *factor strategy models* [Mar02, páginas 109–130]. Se analizan dos versiones de una aplicación de negocios de gran tamaño relacionada con la asistencia informática a la planificación de rutas. La primera versión está compuesta por 93.000 líneas de código, 18 paquetes, 152 clases y 1.284 métodos. La segunda versión, que contiene 115.600 líneas de código, 29 paquetes, 387 clases y 3.446 métodos, ha aumentado en funcionalidad y al mismo tiempo ha sido rediseñada para eliminar algunos *design smells* detectados en la primera versión. El autor sentencia que el uso de los *factor strategy models* muestra cómo la mantenibilidad del sistema se ha mejorado de forma significativa entre las dos versiones del sistema.

En [LWN07a], Lozano *et al.* argumentan que, con el fin de evaluar el impacto de los *design smells* sobre la mantenibilidad de un sistema, debe de ser analizada la evolución del sistema así como de los propios *design smells* detectados entre las diferentes versiones del sistema. Sobre la base de esta premisa, los mismos autores presentan en [LWN07b] un estudio de casos sobre el *design smell Duplicated Code* (en realidad, ellos emplean el término *code clones*). Este trabajo experimental preliminar revela que los clones introducen dependencias implícitas en el código, provocando un aumento de los esfuerzos requeridos para el mantenimiento del sistema y la aparición de otros *design smells* como *Shotgun Surgery*.

Ratzinger *et al.* describen en [RFG05] cómo llevan a cabo con éxito un estudio de casos para detectar y corregir *design smells* en un sistema de distribución y archivo de imágenes. En este artículo, explican como la corrección de *design smells* relacionados con acoplamiento de cambios conduce a una mejora en la capacidad de evolución del sistema.

Otros autores han tratado de explorar en qué situaciones un *bad smell* supone un problema de diseño y en cuáles no. Incluso, a pesar de que la presencia de código duplicado se considera un defecto de diseño que afecta negativamente a la mantenibilidad del sistema, –Fowler *et al.* lo mencionan como “el número uno en la lista del hedor” [FBB⁺99, página 76]–, existen algunas situaciones en las que el empleo de código duplicado puede ser considerado como una buena decisión de diseño. En [KG06], Kasper *et al.* se intentan recopilar ocho patrones de uso de código duplicado (ellos emplean el término *code clones*) para analizar en qué casos es realmente un problema. De este modo, describen los casos particulares en los que no sólo el código duplicado no es un problema, sino que puede ser la mejor opción de diseño. Como resultado, este trabajo es una advertencia para las prácticas de detección de *design smells*, que también sirve para confirmar lo acertado del término *smell* a la hora de referirse a este tipo de problemas de diseño.

No todos los *design smells* constituyen en realmente problemas de diseño. Un procedimiento automático para la gestión de *design smells* debe proporcionar un modo para poder diferenciar estos casos, bien sea de forma automática o con la intervención del desarrollador.

Las mejoras obtenidas mediante la aplicación de refactorizaciones en general, sin estar específicamente dirigidas a la corrección de *design smells*, han sido estudiadas y evaluadas por otros autores. Kataoka *et al.* describen en [KIAF02] un método para evaluar el efecto de las refactorizaciones sobre la mantenibilidad de un sistema, mediante el cálculo de métricas de acoplamiento. El estudio de casos presentado en este trabajo muestra cómo la aplicación de ciertas refactorizaciones mejora la mantenibilidad del sistema.

Moser *et al.* presentan en [MAP⁺08] un estudio de casos dirigido a comprobar el impacto del uso de refactorizaciones en un entorno similar a un entorno de producción real. El sistema analizado es un producto software comercial para monitorizar aplicaciones de dispositivos móviles JAVA. El proyecto software se llevó a cabo en un entorno de desarrollo ágil, y mediante un equipo de desarrollo formado por desarrolladores profesionales y estudiantes. Los resultados indican que el uso de refactorizaciones no sólo mejora aspectos de calidad del software, sino que también incrementa la productividad.

Stroggylos *et al.* analizan en [SS07] si las tareas de refactorización están siendo efectivamente utilizadas dentro de la comunidad de desarrollo de software libre para mejorar la calidad del software. Con este fin, evalúan los registros de los sistemas de control de versiones donde se alojan algunos sistemas de software libre populares, como APACHE HTTPD, APACHE LOG4J, MYSQL CONNECTOR/J y JBOSS HIBERNATE con el objetivo de detectar cambios marcados como refactorizaciones. Examinaron asimismo cómo el uso de prácticas de refactorización afectaba a las métricas. Los autores concluyeron que no todos los procesos de refactorización desembocaban en mejoras en el diseño y en la calidad de los sistemas. Al contrario, los resultados obtenidos mostraban cómo algunas métricas como LCOM³ pueden verse incrementadas, indicando un empeoramiento del diseño, si los desarrolladores no aplicaban las refactorizaciones de una forma apropiada.

Neil *et al.* argumentan en [NL06] que para que el uso de refactorizaciones sea satisfactorio estas deben planificarse con anticipación y teniendo en mente el objetivo de la mejora del diseño. Los autores sugieren seguir un enfoque que denominan “refactorización estratégica” y que describen como “refactorizaciones basadas en patrones de diseño con consciencia de la arquitectura”. Esta técnica recomienda planificar las refactorizaciones a través de la corrección de defectos de diseño y la aplicación de patrones de diseño. También indican que, según sus estudios, la refactorización estratégica basada en patrones de diseño es el método más efectivo para atajar el fenómeno de deterioro de un sistema en orientación al objeto. Los autores analizan un estudio de casos en el que esta propuesta ha sido aplicada con éxito, pero advierten que el diseño puede incluso deteriorarse más, si no se aplican las refactorizaciones correctas.

³Lack of cohesion of methods (falta de cohesión en los métodos) [CK91].

1.2. Tesis, objetivos y contribuciones

Este trabajo se ha estructurado en torno al siguiente enunciado de tesis:

La actividad de refactorización, cuando se requiere aplicar secuencias complejas de refactorizaciones, como en el caso de la corrección de design smells en el software orientado a objetos, puede ser asistida mediante planes de refactorización que pueden ser obtenidos de forma automática.

Para abordar este enunciado de tesis, se ha desglosado en las siguientes hipótesis:

- El estado actual de la gestión de *design smells* es maduro en el caso de la detección, pero todavía debe ser mejorado en el caso de la corrección.
- Las sugerencias de refactorización producidas por las herramientas actuales no son aplicables directamente.
- La corrección de *design smells* mediante refactorizaciones se corresponde con el esquema general de la aplicación de secuencias complejas de refactorización con un objetivo estratégico.
- Las refactorizaciones complejas pueden ser facilitadas gracias a la planificación de refactorizaciones, habilitando las precondiciones de estas mediante refactorizaciones preparatorias que pueden ser obtenidas de forma automática.
- La corrección de *design smells*, como un tipo especial de proceso complejo de refactorización, puede ser facilitado por medio de la planificación de refactorizaciones.

1.2.1. Objetivos

Este trabajo se ha enfocado a los siguientes objetivos:

1. Proporcionar un soporte automático o semi-automático para planificar con antelación las secuencias de refactorizaciones preparatorias –planes de refactorización– que permitan habilitar las precondiciones de un conjunto de refactorizaciones deseado.
2. Proporcionar un soporte automático o semi-automático para respaldar la generación de secuencias de refactorizaciones –planes de refactorización–, que permitan transformar un sistema, según una propuesta de rediseño, mientras se preserva el comportamiento del mismo. Más concretamente, proporcionar un soporte automático o semi-automático para la generación de planes de refactorización orientados a la corrección de *design smells*.
3. Proporcionar un modo para facilitar a los desarrolladores el uso de las técnicas elaboradas en esta tesis.
4. Evaluar la efectividad, eficiencia y escalabilidad de la propuesta presentada en esta tesis mediante el desarrollo de un prototipo que implemente esta propuesta y la realización de un estudio experimental con el mismo.

1.2.2. Resumen de contribuciones

Esta tesis doctoral presenta las siguientes contribuciones:

- Una revisión de la literatura relativa a *design smells*, una descripción histórica de los distintos enfoques de gestión de *design smells* y una propuesta terminológica dirigida a clarificar y a unificar los términos y conceptos relacionados con este problema.
- Una revisión exhaustiva del estado del arte de la gestión de *design smells* y una taxonomía basada en modelos de características, todo ello realizado junto con otros autores, para caracterizar los enfoques y las herramientas actuales y futuras.
- La definición del concepto de *estrategias de refactorización* como un modo de escribir especificaciones automatizables de procesos complejos de refactorización.
- La definición de un lenguaje de especificación de estrategias de refactorización para que los desarrolladores de software puedan emplearlo para escribir estrategias para la corrección de *design smells* y otros procesos complejos de refactorización.
- La definición del concepto de *planes de refactorización* como secuencias de refactorización específicas, instanciadas a partir de estrategias de refactorización, que pueden ser aplicadas de manera efectiva sobre un sistema, dado su estado actual.
- La definición de los requisitos que debe cumplir un enfoque para poder ser utilizado en la computación de planes de refactorización.
- Una técnica para instanciar estrategias de refactorización en planes de refactorización por medio de planificación automática.
- Una línea base y un prototipo de referencia para la investigación futura en planificación automática de refactorizaciones.

1.3. Estructura de la tesis

La presente tesis doctoral se ha organizado como se especifica a continuación.

En el capítulo 2 se presenta el contexto del problema que motiva esta tesis. Se describe qué son *design smells*, se propone una terminología para unificar y referirse de forma homogénea a este tipo de problemas y se realiza también un recorrido histórico sobre los *design smells*. El capítulo también incluye una breve introducción a la refactorización en general y a la automatización de refactorizaciones en particular, y describe la relación entre los *design smells* y las refactorizaciones. Finalmente se argumenta cómo la necesidad de mejorar la actividad de la corrección de *design smells* puede conducir a la mejora de la automatización de procesos complejos de refactorización.

En el capítulo 3 se analiza el estado del arte de la gestión de *design smells* y se realiza una revisión exhaustiva de los distintos enfoques y herramientas relacionados con la detección, la corrección, la visualización, etc. de *design smells*. Para realizar este análisis se define una taxonomía basada en modelos de características que sirve para identificar, caracterizar y comparar los distintos enfoques existentes y los que puedan aparecer en el futuro.

En el capítulo 4 se definen los conceptos de estrategias de refactorización y planes de refactorización. En primer lugar, en este capítulo se analiza cómo se soporta actualmente la corrección

de *design smells* mediante la especificación de heurísticas de corrección. Después se presentan las estrategias de refactorización y los planes de refactorización como medios para escribir especificaciones más formales para los procesos complejos de refactorización, que pueden fomentar la automatización de la actividad de corrección de *design smells*. Finalmente, en este capítulo se describen las características del problema de la automatización de la instanciación de estrategias de refactorización en planes de refactorización.

En el capítulo 5 se discute cuáles son las técnicas más convenientes para implementar la instanciación automática de estrategias de refactorización, y se presenta la técnica seleccionada, la planificación de redes jerárquicas de tareas. Se incluye una breve introducción a la planificación automática y a la planificación de redes jerárquicas de tareas y el contenido restante de este capítulo se dedica a realizar un análisis sobre cómo las estrategias de refactorización pueden traducirse y automatizarse mediante esta técnica.

En el capítulo 6 se describe un estudio de casos que valida la propuesta presentada en esta tesis. Las especificaciones para corregir dos *design smells*—*Feature Envy* y *Data Class*— se recopilan en forma de estrategias de refactorización y después son traducidas a un dominio de planificación de refactorizaciones, con el fin de automatizarlas. El prototipo de planificador de refactorizaciones desarrollado en esta tesis se ejecuta sobre un conjunto de sistemas de software libre que presenta numerosos *design smells*, con el objetivo de obtener los planes de refactorización apropiados para eliminar dichos *design smells*. En este capítulo se describen estos experimentos y se discuten sus resultados. Finalmente, la propuesta para la corrección de *design smells* presentada en esta tesis se caracteriza usando la taxonomía definida en el capítulo 3.

En el capítulo 7 se resumen las conclusiones de esta tesis, se discuten sus resultados, contribuciones y limitaciones, y se presentan las perspectivas de investigación abiertas.

Capítulo 2

Contexto

Este capítulo presenta un recorrido histórico sobre la investigación en gestión de problemas de diseño en software orientado a objetos. Se propone una terminología unificada para hacer referencia a este tipo de problemas como *design smells*, y se identifica la corrección automática de *design smells* como el siguiente hito a abordar en este campo.

En este capítulo se relata cómo se han utilizado diferentes nombres para hacer referencia a problemas de diseño similares. Con el fin de facilitar la comunicación entre los investigadores de este campo, en este capítulo proponemos una terminología común para estos problemas. En concreto, proponemos el término *design smell* con la siguiente definición:

Definition 1. Un **design smell** es un problema que se presenta en la estructura del software (ya sea en el código o en el diseño), que no provoca errores en tiempo de compilación ni en tiempo de ejecución, pero que afecta negativamente a los factores de calidad del software.

Teniendo en cuenta las diversas definiciones de *design smells* que pueden encontrarse en la literatura, se propone además una pequeña clasificación para describir los distintos tipos de *design smells*:

- **Low-Level Smells:** *design smells* de bajo nivel, que pueden ser detectados e identificados por sus propias características.
- **High-Level Smells:** *design smells* de alto nivel, que pueden estar compuestos por otros. En la mayoría de los casos pueden ser descritos y detectados como una confluencia de otros *design smells*, que pueden ser a su vez *high-level smells* o *low-level smells*.

A partir de la revisión de la literatura relativa a *design smells*, se ha elaborado un resumen histórico de la investigación en este campo, que se ha presentado de forma gráfica con una sencilla figura (ver figura 2.1). Esta figura destaca algunos de los trabajos que consideramos clave en cada avance del estado del arte. Desde los primeros estudios, en los que se formulaban pautas generales de buenas prácticas de diseño orientado a objetos [Rie96], el campo ha crecido y madurado para ofrecer enfoques más automatizados, no sólo para la detección de *design smells*, sino también en cuanto a todas las demás actividades relacionadas con la gestión de *design smells* como, por ejemplo, la especificación, la corrección, etc. En relación con la corrección y la detección, el campo se ha desarrollado proporcionando especificaciones cada vez más detalladas y formales. Esto ha tenido como consecuencia una mejora en la automatización de ambas actividades.

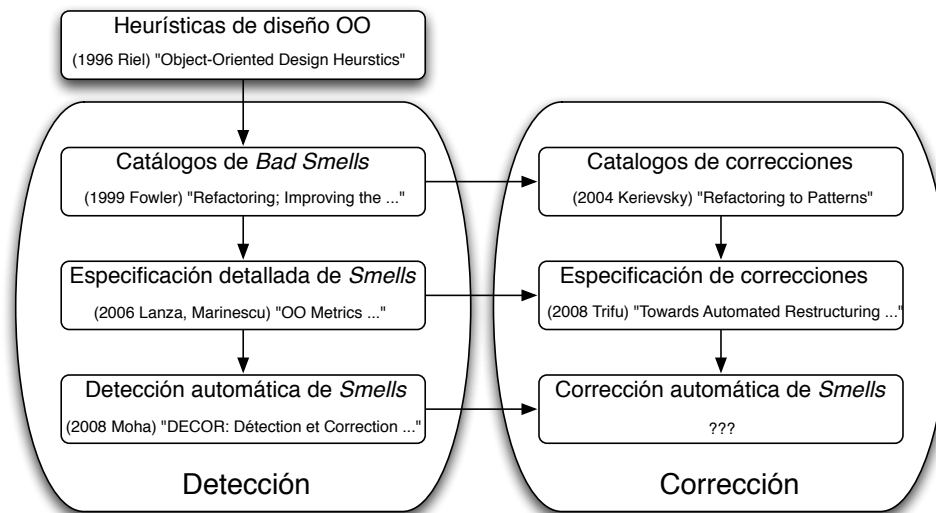


Figura 2.1: Breve resumen histórico sobre la gestión de design smells.

Tras analizar la situación actual de la investigación en este campo, se concluye que la actividad de detección ha alcanzado un grado de madurez que hace prever que puedan aparecer en breve herramientas de desarrollo que ofrezcan estas funcionalidades en entornos reales de producción. Por otra parte, se ha identificado la automatización de la corrección de *design smells* como el siguiente hito que se debe alcanzar.

En este capítulo se realiza también una breve introducción al concepto de “refactorización” [Opd92]: transformaciones estructurales que pueden aplicarse al código fuente de un sistema software con el propósito de realizar cambios en el diseño sin modificar el comportamiento observable.

Se realiza una revisión de los distintos hitos en la investigación en el campo de las refactorizaciones según los diferentes escenarios en los que se puede emplear esta técnica:

- **aplicación de refactorizaciones:** asistencia a la aplicación de refactorizaciones de forma automática o semi-automática mediante entornos de desarrollo integrados o herramientas de refactorización.
- **desarrollo de herramientas de refactorización:** asistencia al desarrollo de herramientas de refactorización, con el fin de facilitar la construcción de mejores herramientas, que soporten más tipos de refactorizaciones, más lenguajes de programación y permitan aplicar refactorizaciones de forma más segura y sencilla.
- **minería de refactorizaciones:** análisis de diferentes versiones de un sistema software para averiguar las refactorizaciones que se hayan podido aplicar entre las diferentes versiones, con el fin de mejorar la comprensión de la evolución del sistema.
- **sugerencias de refactorización:** generación de propuestas de refactorizaciones que podrían aplicarse sobre un sistema con el fin de conseguir un objetivo determinado como, por ejemplo, la mejora de un determinado factor de calidad.

- **corrección de *design smells*:** uso de refactorizaciones para llevar a cabo la actividad de corrección de *design smells*.

Finalmente se profundiza y se describen las relaciones entre refactorizaciones y *design smells*, y se argumenta asimismo que las operaciones de refactorización son la técnica más apropiada para poder introducir en un sistema los cambios necesarios para la corrección de *design smells*, según se establece también en [FBB⁺99].

Capítulo 3

Estudio exhaustivo de la gestión de *design smells*

En este capítulo se realiza una revisión del estado del arte de la gestión de *design smells* y se presenta un estudio exhaustivo de los enfoques y herramientas de *design smells* existentes. Mediante el uso de diagramas de características se ilustra de forma gráfica el estudio y se define una taxonomía. Este estudio puede ser empleado con varios propósitos. Los recién llegados al campo pueden utilizarlo para llegar a adquirir y conocer los aspectos más importantes de la gestión de *design smells*, los desarrolladores de herramientas pueden utilizarla para comparar y mejorar sus herramientas, y los desarrolladores de software pueden utilizarla para evaluar qué herramienta o técnica es la más apropiada para sus necesidades.

Este estudio se ha elaborado en colaboración con Naouel Moha, Tom Mens y Carlos López. Una versión previa del estudio y de la taxonomía presentada en esta tesis se encuentra disponible como informe técnico [PLMM11]. También se ha realizado una caracterización detallada de un conjunto de herramientas de gestión de *design smells*, según dicha versión previa de la taxonomía [MASFPC11]. Los resultados de esta caracterización también se han hecho públicos en un portal web ¹.

Como guía para describir nuestro estudio, hemos empleado una notación visual denominada “diagramas de características”, inspirándonos en cómo han utilizado esta notación Czarnecki y Helsen para presentar un estudio sobre transformación de modelos [CH06]. Los modelos de características [CE00] permiten representar las propiedades comunes y variables de diferentes conceptos y las dependencias entre ellos. Este modelo se utiliza para representar jerarquías de características representando las propiedades comunes y variables de las instancias de los conceptos representados y las dependencias entre las características variables.

La característica principal de nuestra taxonomía, la raíz del modelo, es la *Gestión de Design Smells*. Esta representa cualquier enfoque que se ocupe de la gestión de *design smells*. Una herramienta de gestión de *design smells* se corresponderá con una instancia de este modelo de características en la que se implementan todas las características obligatorias y se seleccionan algunas de entre las disponibles en los puntos de variabilidad identificados. En la figura 3.1 se muestra la característica principal de la taxonomía y sus subcaracterísticas. Estas representan las dimensiones de variabilidad principales en los enfoques de gestión de *design smells*. Todas las herramientas de gestión de *design smells* tienen que implementar estas características.

¹<http://www.infor.uva.es/DesignSmells>

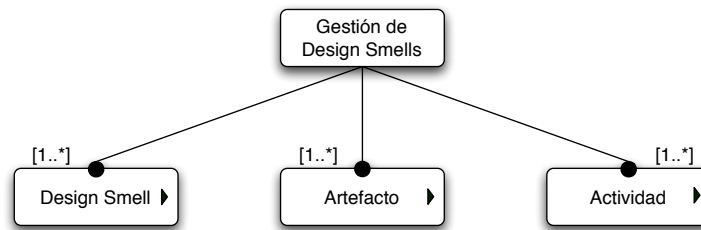


Figura 3.1: Raíz de la taxonomía, puntos de variabilidad principales para comparar los distintos enfoques de design smells.

Las características principales identificadas son:

- **Design Smell:** El tipo y naturaleza de los *design smells* gestionados por cada herramienta o enfoque. Cualquier enfoque debe tratar al menos un tipo de *design smell*.
- **Artefacto:** Todas las herramientas o técnicas deben tratar *design smells* que aparecen en determinados tipos de artefactos software. Esta característica identifica los artefactos que pueden ser tratados.
- **Actividad:** El tipo de actividad de gestión de *design smells* que se soporta mediante una determinada técnica o herramienta. Todas las herramientas existentes deben ofrecer soporte para al menos una actividad.

En la presente tesis se profundiza en la clasificación según las tres dimensiones principales definidas en la figura 3.1. En este resumen nos centraremos sólo en las subcaracterísticas de la característica *Actividad*.

La característica *Actividad* representa una dimensión principal de variación entre los diferentes enfoques de gestión de *design smells* existentes. En esta tesis, hemos identificado que el proceso de gestión de *design smells* puede dividirse en 5 tipos de actividades: *Especificación*, *Detección*, *Visualización*, *Corrección* y *Análisis del Impacto*. Sus subcaracterísticas (no se muestran en este resumen) profundizan en cómo se soporta cada tipo de actividad. En concreto, se examinan tres subcaracterísticas: *Técnica*, las técnicas empleadas para asistir esa actividad; *Automatización*, el grado de automatización conseguido para esa actividad; y *Resultado*, la naturaleza de los resultados producidos para esa actividad.

En la figura 3.2 se muestra tanto la característica *Actividad* y los diferentes tipos de actividades de gestión de *design smells* como sus características. Estas se enumeran a continuación:

- **Especificación:** La actividad se refiere a si se proporciona o no a los desarrolladores la ayuda necesaria para ampliar o adaptar la herramienta o la técnica a sus necesidades particulares, permitiendo la especificación de nuevos *design smells* o modificando las especificaciones ya existentes.
- **Detección:** La actividad se refiere a si la técnica o herramienta ofrece la posibilidad de detectar *design smells*.
- **Visualización:** Esta actividad se refiere a las técnicas o herramientas que produce una cierta clase de representación gráfica del artefacto, permitiendo la identificación rápida y

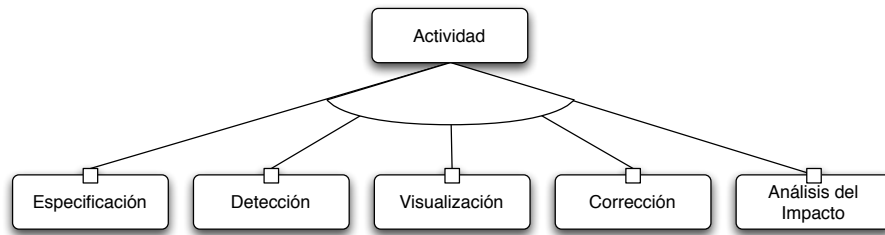


Figura 3.2: Característica actividad y sus subcaracterísticas.

sencilla de algunas de sus propiedades. Los enfoques o herramientas que permiten realizar esta actividad en el contexto de la gestión de *design smells* pueden ofrecer la posibilidad de realizar otras actividades de forma visual como, por ejemplo, facilitar la detección de *design smells*, ayudar a decidir cuáles son las mejores modificaciones para eliminar un *design smell*, comprender las causas y el impacto de determinados *design smells*, etc.

- **Corrección:** La implementación de esta actividad se refiere a si se ofrece o no un soporte para la corrección de *design smells*, ya sea proporcionando sugerencias de modificación del artefacto, generando programas que aplicaran las transformaciones necesarias o incluso modificando automáticamente el artefacto.
- **Análisis del Impacto:** Esta actividad se refiere a la capacidad de una técnica o herramienta para averiguar el impacto que tiene la presencia de un *design smell* en el sistema, o bien el impacto de las transformaciones que se deberían aplicar para eliminarlo.

De entre las diferentes conclusiones que se han podido extraer a partir de este estudio, a modo de resumen, cabe destacar las siguientes.

Hemos observado que no son muchos los enfoques estudiados que permiten razonar sobre *design smells* a nivel de modelos, en comparación con la cantidad de trabajos que se refieren a código fuente y código ejecutable. Sería deseable avanzar en esta línea para ofrecer una mejora en la gestión de *design smells* a nivel de modelos.

Durante nuestro estudio, hemos podido constatar que existen considerablemente menos enfoques que soportan la corrección automática de *design smells*, que los que ofrecen la detección o la visualización de *design smells*. Es más, mientras que las herramientas de detección se encuentran en un estado de desarrollo bastante maduro, las herramientas que soportan la actividad de corrección son principalmente prototipos de investigación. La generación siguiente de herramientas de gestión de *design smells* debe dirigirse a mejorar la integración y automatización de las técnicas de corrección.

Capítulo 4

Estrategias de refactorización

En este capítulo se revisa la situación actual en la corrección de *design smells* y se realiza un análisis de este dominio. Se presentan los modelos que describen cómo se propone en esta tesis que se aborde el problema y cómo puede mejorarse la actividad de corrección mediante el cálculo de secuencias complejas de refactorizaciones. Se definen las estrategias de refactorización como un concepto que permite abordar el problema de planificar secuencias complejas de refactorizaciones con antelación. Finalmente, se presentan los requisitos que debe cumplir una técnica para soportar el enfoque de corrección de *design smells* basado en estrategias de refactorización y se identifican las características de este problema.

Se han revisado una serie de trabajos relacionados con la corrección de *design smells* mediante refactorizaciones: el libro de antipatrones de Brown *et al.* [BMMIM98], el catálogo de *bad smells* de Fowler *et al.* [BF99], los ejemplos de refactorizaciones del libro de Wake [Wak03], las refactorizaciones relacionadas con patrones de diseño del libro de Kerievsky [Ker04], las *disharmonies* (“disonancias” en el diseño) del libro de Lanza y Marinescu [LM06], el catálogo de los modelos de reingeniería de Demeyer *et al.* [DDN08] y las estrategias de corrección de la tesis doctoral de Trifu [Tri08]. Para ilustrar el resultado de este análisis del dominio, hemos seleccionado el mismo *design smell* –*Large Class*– con el fin de poder comparar los diferentes estilos y nivel de detalle en las diferentes estrategias de corrección para este *design smell* que describen estos autores.

A partir de este análisis, se ha obtenido un modelo general que representa cómo es una estrategia de corrección de *design smells* desde el punto de vista de todos estos diferentes enfoques. Con el fin de integrar las especificaciones de estas estrategias con las especificaciones de refactorizaciones, también se ha elaborado un modelo para estas. Además a partir de este análisis, se han enunciado los problemas que se deben abordar para mejorar la actividad de corrección de *design smells* mediante la aplicación de refactorizaciones:

- **Aplicabilidad de las refactorizaciones:** El incumplimiento de las precondiciones dificulta la aplicación de refactorizaciones. Se necesita una técnica que permita calcular qué secuencias de refactorizaciones se pueden aplicar para habilitar las precondiciones de la refactorización que se deseaba aplicar originalmente.
- **Especificaciones de corrección poco formales:** Las estrategias de corrección de *design smells* se describen de forma heurística a partir del conocimiento empírico recabado por los desarrolladores de software. Se necesita una técnica que permita especificar estas “recetas” de corrección de un modo más formal y que permita automatizar su aplicación.

Para abordar los problemas identificados se propone generalizar los enfoques de corrección de *design smells* en un enfoque más amplio. Para ello se proponen los conceptos de *Estrategias de refactorización* y *Planes de refactorización* que se definen del siguiente modo:

- **Estrategias de refacotrización:** Especificaciones de transformaciones complejas de software que persiguen un determinado objetivo de diseño, que preservan el comportamiento del sistema, están basadas en heurísticas, pueden ser automatizadas y pueden ser instanciadas, para cada caso particular, en planes de refactorización
- **Planes de refactorización:** Secuencias de instancias de transformaciones que persiguen un objetivo de diseño determinado, que pueden ser aplicadas directamente sobre un sistema dado su estado actual, que preservan el comportamiento del sistema y que pueden ser instanciadas a partir de estrategias de refactorización.

Por lo tanto, las estrategias de refactorización representan una manera de especificar de forma organizada y estructurada el conocimiento empírico y las heurísticas de aplicación de secuencias de refactorizaciones complejas, mientras que los planes de refactorización representan dichas secuencias y se obtienen como instancias de las estrategias de refactorización para un caso particular.

A continuación, en este capítulo se definen en detalle estos conceptos mediante modelos representados con diagramas de clases UML. Se define también un pequeño lenguaje específico de dominio para la especificación de estrategias de refactorización y se presentan algunos ejemplos acerca de cómo se formalizarían las estrategias de corrección para el *design smell Large Class* mediante este lenguaje.

Finalmente, a modo de conclusiones del capítulo, se describen las características del problema de instanciación de estrategias de refactorización y de generación de planes de refactorización que se recogen a continuación brevemente.

El soporte que se emplee para instanciar estrategias de refactorización deberá permitir:

- **Computación:** calcular los conflictos y dependencias entre diferentes refactorizaciones a partir de su especificación, o bien permitir calcular los efectos de la ejecución de una refactorización mediante la simulación de su aplicación.
- **Representación del sistema:** representar un sistema software al nivel de detalle del código fuente, por ejemplo mediante una representación basada en árboles de sintaxis abstracta (ASTs). También debe permitir la manipulación de este modelo al nivel de sus elementos más simples.
- **Estructuras de control deterministas y no-deterministas:** describir transformaciones de forma algorítmica mediante estructuras de control deterministas y no-deterministas.
- **Especificaciones incompletas:** obtener planes de refactorización, aún cuando el conocimiento empírico recogido en las estrategias de refactorización no sea todo lo exhaustivo que se pudiera desear.
- **Elementos de estrategias de refactorización:** implementar todos los elementos que se describen en los modelos que se han empleado para definir en detalle los conceptos de estrategias y planes de refactorización.

Capítulo 5

Planificación de refactorizaciones

Este capítulo se ha centrado en la descripción de la técnica que en esta tesis proponemos utilizar para soportar la automatización de estrategias de refactorización y la planificación de refactorizaciones. En este capítulo argumentamos que el problema de planificación de refactorizaciones para la corrección de *design smells* puede ser modelado y abordado como un problema de planificación automática [GNT04]. En concreto, en este capítulo se describe cómo se aborda el problema mediante planificación de redes jerárquicas de tareas (**Hierarchical Task Network (HTN) planning**) [GNT04, Capítulo 11].

Mediante este enfoque la mayoría de los problemas del dominio de planificación de refactorizaciones son problemas típicos ya tratados en el campo de la planificación automática. Por otra parte, los detalles particulares necesarios para la generación de planes de refactorización se recopilan como conocimiento del dominio que se le proporciona al planificador para que efectúe la búsqueda de las secuencias de refactorizaciones deseadas.

Por consiguiente, el conocimiento empírico acerca de la corrección de *design smells* se implementará y refinará de forma incremental. Una vez diseñado el prototipo del planificador, la mayor parte de los esfuerzos de desarrollo se han de dedicar a la elaboración de un dominio de planificación de refactorizaciones. Este dominio estará compuesto por las heurísticas de corrección de *design smells* y de aplicación de refactorizaciones que se hayan formalizado previamente como estrategias de refactorización.

Este capítulo incluye una breve introducción a la planificación automática, una argumentación sobre el enfoque de planificación automatizada que se ha seleccionado en particular –*HTN planning*– y una formalización del problema de planificación de refactorizaciones mediante el planificador JSHOP2, perteneciente a la familia de planificadores seleccionada.

Capítulo 6

Estudio de casos

Con el fin de demostrar la viabilidad de nuestra propuesta, hemos desarrollado un pequeño dominio HTN para el problema de planificación de refactorizaciones que incluye estrategias para la corrección de los *design smells* *Feature Envy* y *Data Class*. Se ha contruido también un prototipo que integra herramientas de otros investigadores y se ha llevado a cabo un estudio de casos sobre un conjunto de sistemas de código abierto, con el fin de caracterizar este prototipo. En este capítulo se ha incluido una descripción del dominio de planificación de refactorizaciones que hemos elaborado, el prototipo de referencia que hemos construido, y un análisis y discusión de los resultados obtenidos con ellos para los dos tipos de *design smells* abordados. Una vez que, junto con la información contenida en este capítulo, ya se dispone de la definición completa de la técnica desarrollada en esta tesis, se caracteriza la propuesta de corrección de *design smells*, que se presenta en esta tesis mediante la taxonomía elaborada en el capítulo 3. En el presente resumen de la tesis, la caracterización puede consultarse en la sección 7.4 del capítulo de conclusiones (capítulo 7).

Nuestra principal contribución en este prototipo ha sido el dominio de planificación de refactorizaciones que hemos elaborado para demostrar nuestra propuesta. Con el fin de dotar del conocimiento necesario al planificador JSHOP2, hemos incorporado las especificaciones de un conjunto de refactorizaciones, las estrategias de refactorización y otras transformaciones y consultas del sistema, como redes de tareas HTN escritas en el lenguaje del planificador. En la práctica, esto significa en realidad programar estas transformaciones y consultas en el lenguaje de definición dominios de JSHOP2. Escribir y depurar el dominio de planificación de refactorizaciones ha sido la actividad más compleja en la implementación de nuestra propuesta. Sin embargo, estas especificaciones se pueden reutilizar fácilmente.

Para validar nuestra propuesta, hemos escrito estrategias de refactorización para los *design smells* *Remove Data Class* y *Remove Feature Envy*; y para la refactorización **MOVE METHOD**. Del mismo modo, se han escrito las especificaciones para 9 operaciones de refactorización: **ENCAPSULATE FIELD**, **MOVE METHOD**, **RENAME METHOD**, **RENAME FIELD**, **RENAME PARAMETER**, **RENAME LOCAL VARIABLE**, **REMOVE FIELD**, **REMOVE METHOD** y **REMOVE CLASS**. También hemos utilizado la combinación de herramientas ECLIPSE + JTRANSFORMER para desarrollar y depurar un conjunto de consultas del sistema, más de 150, que almacenamos en un fichero PROLOG.

Como ejemplo de las estrategias de corrección implementadas, se muestra en la figura 6.1 la representación de una red de tareas HTN que implementa un conjunto de estrategias de refactorización para corregir el *design smell* *Data Class*.

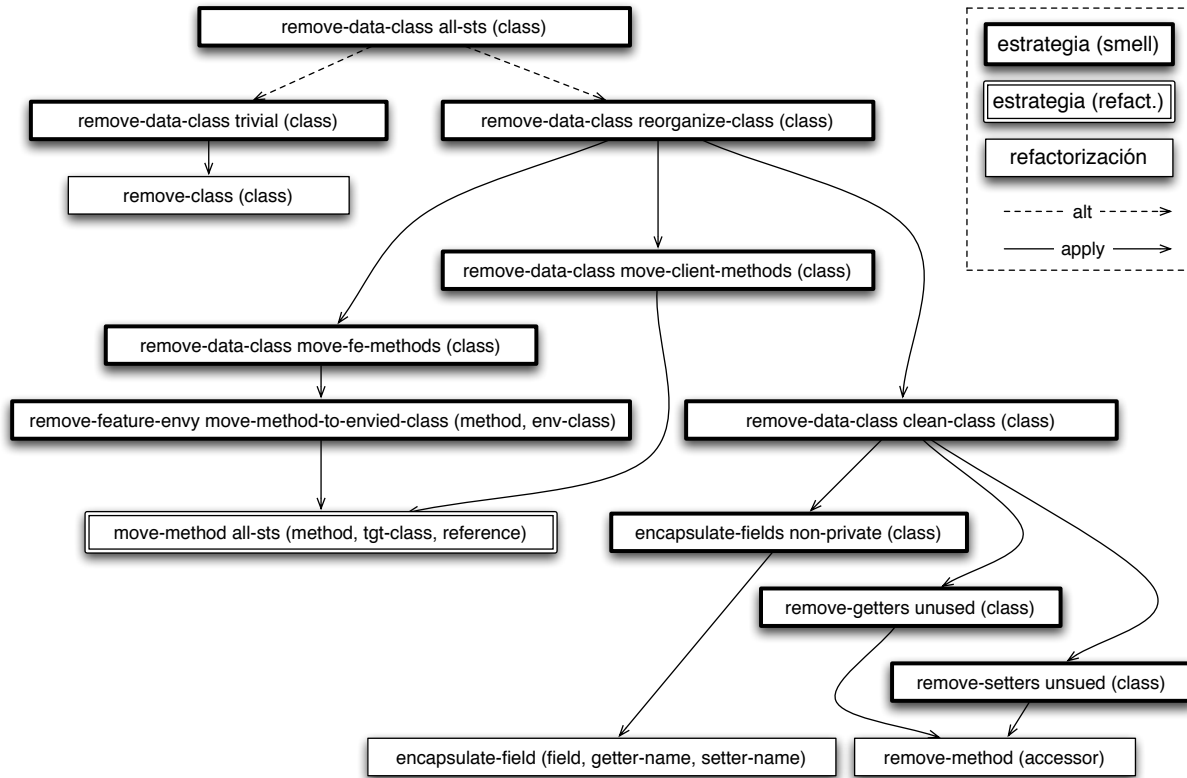


Figura 6.1: Visión general de un conjunto de estrategias sencillas para eliminar una Data Class.

Para poder realizar las pruebas necesarias y validar el dominio de planificación de refactorizaciones elaborado se ha construido un prototipo que integra un conjunto de herramientas de otros autores. Dichas herramientas son: JTRANSFORMER [JTr], JSHOP2 [JSH] e IPLASMA [iPl]. Cabe señalar que nuestra intención ha sido demostrar el enfoque presentado en esta tesis y no construir una herramienta que pueda emplearse en un entorno de producción. Por lo tanto, en algunos casos la facilidad de integración con fines de experimentación, y la disponibilidad han sido los criterios que hemos seguido al seleccionar una herramienta o una versión particular de entre las diferentes variantes existentes. A modo de resumen, el esquema general de las herramientas integradas y de los diferentes ficheros que conforman nuestro prototipo se presenta en la figura 6.2.

Para estudiar el prototipo y evaluar nuestra propuesta hemos realizado algunos experimentos basados en estudios de casos. Se han presentado dos estudios orientados a instanciar planes de refactorización a partir de estrategias de refactorización para eliminar los *design smells* Data Class y Feature Envy. Para el estudio se ha planteado el siguiente objetivo:

Objetivo del experimento: Analizar el enfoque de planificación de refactorizaciones con el propósito de caracterizarlo y evaluarlo con respecto a su eficacia, eficiencia y escalabilidad desde el punto de vista del investigador en el marco de referencia del prototipo que hemos construido.

	Sistema	Versión	NOC	NOM	LOC	PEF	Feature Envy	Data Class
1	JTombstone	1.1.1	40	233	1938	32780	2 (2)	7
2	Groom	1.3	34	243	3699	35434	2 (5)	2
3	Lucene	1.9	199	1998	17627	85064	18 (25)	16
4	Pounder	0.96	218	1318	9410	98570	26 (26)	3
5	MyTelly	1.2	72	941	12625	133605	2 (6)	13
6	Jwebap	0.6.1	114	1278	16417	141047	17 (18)	21
7	dbXML	2.0	389	3336	25862	199400	21 (30)	40
8	GanttProject	2.0.10	515	5048	40775	241095	36 (64)	27
9	JfreeChart	1.0.11	561	8024	80668	354543	45 (46)	29
Número total de experimentos							169	158

Cuadro 6.1: Detalles de los sistemas utilizados en nuestros experimentos. Se indica el número de clases (NOC), de métodos (NOM), de líneas de código (LOC), de predicados que conforman la representación del sistema (PEF) y el número de manifestaciones de los design smells Data Class y Feature Envy.

Como resumen final del estudio, los análisis estadísticos realizados a partir de los resultados de los experimentos han conducido a las siguientes conclusiones:

■ **Eficacia:**

- La eficacia de nuestra propuesta, en términos de los planes de refactorización producidos para cada caso ha sido bastante satisfactoria —el 42 % de los casos de *Feature Envy* y el 92 % de los casos de *Data Class*—, sobre todo teniendo en cuenta la relativa simplicidad del dominio de planificación de refactorizaciones implementado.

■ **Eficiencia y escalabilidad:**

- Existe una correlación evidente entre el tiempo de pre-compilación¹ y el tamaño del sistema.
- El tiempo de planificación no sigue una distribución normal, y por lo tanto hemos utilizado pruebas no paramétricas para todos los análisis estadísticos inferenciales.
- El tiempo de planificación depende del tamaño del sistema, aunque no se ha podido encontrar una función de correlación.
- El tiempo de planificación parece no depender de la estrategia solicitada, aunque no se han obtenido resultados concluyentes a este respecto.
- Las cotas probabilísticas superiores del tiempo de planificación para los sistemas evaluados han resultado muy satisfactorias, especialmente teniendo en cuenta que se trata de un prototipo.
- Se requieren más datos y experimentos, incluyendo más sistemas y más estrategias de refactorización de *design smells*, para poder dar, en un futuro, resultados más concluyentes en cuanto a los tiempos de planificación.

¹El tiempo de pre-compilación pertenece a una fase exclusiva de la ejecución de JSHOP2, que puede eliminarse en próximas versiones del prototipo.

Capítulo 7

Conclusiones

En este capítulo se resumen y discuten los resultados de esta tesis doctoral. También se revisan las diferencias con otros enfoques relacionados, se detallan las principales contribuciones y limitaciones de la propuesta, y se presentan las oportunidades de trabajo futuro y las preguntas que han surgido tras la elaboración de este trabajo.

7.1. Resultados generales

Se ha realizado un estudio histórico sobre las técnicas para mejorar la estructura del software mediante la detección y la corrección de *bad smells* y de otros problemas de diseño relacionados. Se han revisado las semejanzas y las diferencias entre los diversos enfoques y catálogos que abordan problemas de diseño. Como consecuencia, hemos propuesto una terminología unificadora para estos problemas de diseño, para referirnos a ellos de forma homogénea como *design smells*. Se ha analizado la situación actual de la investigación en este campo y se ha identificado la automatización de la corrección de *design smells* como el siguiente hito que se pretende alcanzar.

Se ha llevado a cabo una revisión exhaustiva sobre la gestión de *design smells* y se ha elaborado una taxonomía usando diagramas de características para ilustrarla de forma gráfica. Esta taxonomía puede servir para varios propósitos. Los recién llegados a este campo pueden utilizarla para comprender los aspectos más importantes de la gestión de *design smells*. Los desarrolladores de herramientas de gestión de *design smells* pueden emplearla para comparar y mejorar sus herramientas, mientras que los desarrolladores de software pueden utilizarla para evaluar qué herramienta o técnica es la más apropiada para sus necesidades.

Hemos propuesto una unificación y una generalización de los distintos enfoques existentes para la corrección de *design smells* en forma de estrategias de refactorización y de planes de refactorización. Las estrategias de refactorización permiten recopilar el conocimiento empírico relativo a cómo eliminar un *design smell* o cómo aplicar una refactorización cuando se requieren refactorizaciones preparatorias para habilitar las precondiciones de ésta. Los planes de refactorización se instancian a partir de estrategias de refactorización y representan la secuencia exacta de transformaciones que, preservando el comportamiento del sistema, se pueden aplicar inmediatamente dada la versión actual de su código. Se ha definido y se ha elaborado el concepto de estrategias de refactorización como un modo que permite la descripción y la formalización de procesos complejos de refactorización. Las estrategias de refactorización se han definido en términos de diagramas de clase UML, y se ha propuesto un lenguaje específico de dominio para ellas.

Se han indicado las características principales del problema que conlleva la instanciación de estrategias de refactorización en planes refactorización. Estas características se han utilizado para seleccionar el enfoque más apropiado para abordar la instanciación de planes refactorización, La instanciación de estrategias refactorización se ha representado como un problema de planificación automática. Hemos seleccionado la planificación de redes jerárquicas de tareas (*HTN planning*) y en particular el planificador JSHOP2 como el enfoque y el planificador más apropiados para la instanciación de estrategias refactorización en planes de refactorización.

Hemos concluido que los planes refactorización pueden ser generados con un planificador automático, en concreto, implementando estrategias de refactorización como redes jerárquicas de tareas para un planificador HTN. Para demostrar esto, hemos definido cómo las estrategias de refactorización pueden ser especificadas como redes de tareas en JSHOP2 y cómo el problema de planificación de refactorizaciones se puede abordar como un problema de planificación mediante JSHOP2. Los distintos elementos de las estrategias de refactorización se han formulado como elementos de redes de tareas de JSHOP2. También hemos formulado la instanciación de estrategias de refactorización en planes de refactorización como un problema de planificación en JSHOP2.

Se ha ensamblado un prototipo de ejemplo para demostrar nuestra propuesta. Se han integrado algunas herramientas existentes de otros autores para construir la infraestructura básica del prototipo y se ha desarrollado un dominio de planificación de refactorizaciones: la especificación de un dominio de HTN que incluye consultas sobre el estado del sistema, estrategias de refactorización y especificaciones de refactorizaciones y de otras transformaciones que no preservan el comportamiento.

Se han realizado dos estudios de casos para evaluar nuestra propuesta, y para analizar el rendimiento del prototipo de referencia en términos de eficacia, eficiencia y escalabilidad. Los estudios de casos desarrollados, se han orientado a la eliminación de los *design smells* *Feature Envy* y *Data Class* y se han aplicado sobre 9 sistemas informáticos distintos de pequeño a medio tamaño. Los resultados del estudio confirman que nuestro enfoque se puede utilizar para generar automáticamente planes de refactorización para procesos de refactorización complejos en un tiempo razonable. Los estudios realizados también demuestran que la eficiencia de los planificadores de redes jerárquicas de tareas y la expresividad del lenguaje de especificación de dominios de planificación de JSHOP2, hacen que este sea el planificador más apropiado para soportar el problema de planificación de refactorizaciones.

7.2. Resultados relacionados con el enunciado de la tesis

El enunciado de la tesis, formulado en el Capítulo 1, se reproduce aquí de nuevo:

La actividad de refactorización, cuando se requiere aplicar secuencias complejas de refactorizaciones, como en el caso de la corrección de design smells en el software orientado a objetos, puede ser asistida mediante planes de refactorización que pueden ser obtenidos de forma automática.

La discusión sobre cómo se ha abordado esta tesis se realiza aquí revisando las diferentes hipótesis en las que se ha dividido el enunciado principal. Las declaraciones de la tesis enumeradas en el capítulo 1 se reproducen aquí y se argumentan individualmente:

- Existe madurez en el estado actual de la gestión de *design smells* en el caso de la detección, pero todavía debe ser mejorado en el caso de la corrección.

La revisión del estado del arte realizado en los capítulos 2 y 3 ha servido para verificar esta hipótesis. El grado de automatización alcanzado por las herramientas y enfoques de detección de *design smells* existentes han alcanzado en muchos casos un nivel de automatización completa. Por otra parte, los resultados, según mencionan los autores, son muy aceptables y, como hemos mencionado, creemos que en breve aparecerán herramientas de desarrollo con funcionalidades de detección de *design smells* en entornos de producción.

Sin embargo, la corrección de *design smells* apenas se ha materializado en herramientas. Hay pocos enfoques que hayan abordado la corrección de *design smells*, siendo los catálogos de “recetas” de corrección la referencia principal del estado del arte en esta actividad de la gestión de *design smells*. El grado de madurez alcanzado en general por estas herramientas tan sólo ha permitido alcanzar un nivel de automatización “manual”. Hasta donde sabemos, sólo se han realizado implementaciones de heurísticas de procedimientos de corrección para un conjunto reducido de *design smells* muy específicos. Para estos pocos casos muy especializados si que se ha alcanzado un nivel de automatización “completamente automatizado”. En todo caso, como se menciona en el capítulo 2, en general la actividad de corrección de *design smells* carece de trabajos satisfactorios relacionados con la especificación y la automatización precisa y sistemática.

- Las sugerencias de refactorización producidas por las herramientas actuales no son aplicables directamente.

La revisión del estado del arte realizada en los capítulos 2 y 3 también ha servido para demostrar este punto. Los trabajos que se ocupan de la corrección de *design smells* solamente producen sugerencias de corrección. Estas sugerencias se basan en estrategias generales y son generadas sin tener en cuenta el estado actual del sistema o cómo la sugerencia debe ser aplicada en cada caso particular. Una excepción a esto son algunos enfoques y herramientas que se centran en un sólo *design smell* y que han sido desarrollados para aplicar estrategias específicas para *design smells* muy concretos.

- La corrección de *design smells* mediante refactorizaciones se corresponde con el esquema general de la aplicación de secuencias complejas de refactorización con un objetivo estratégico.

El capítulo 4 se ha dedicado a analizar cómo las sugerencias de corrección de *design smells* se han venido especificado en la literatura hasta ahora. Se ha revisado también cómo se especifican las refactorizaciones complejas y cómo ambos tipos de transformaciones son aplicadas. La comprobación de esta hipótesis se ha abordado elaborando el concepto de estrategias de refactorización, que representan estrategias de corrección de *design smells* en particular, y especificaciones de procesos complejos de refactorización en general. Las estrategias de corrección de *design smells* y las descripciones de refactorizaciones pueden ser descritas de forma homogénea como procesos complejos de refactorización. Por lo tanto, ambos tipos de transformaciones pueden ser especificadas mediante estrategias de refactorización de manera que de ellas se puedan obtener planes de refactorización.

- Las refactorizaciones complejas pueden ser facilitadas gracias a la planificación de refactorizaciones, habilitando las precondiciones de estas mediante refactorizaciones preparatorias que pueden ser obtenidas de forma automática.

Para verificar esta hipótesis, hemos desarrollado un enfoque y un prototipo que emplea planificación de redes jerárquicas de tareas (*HTN planning*) para instanciar planes de refactorización a partir de estrategias de refactorización. Se ha elaborado, como demostración de nuestra propuesta, un dominio de planificación de refactorizaciones para ser utilizado con un planificador. Este dominio contiene, entre otra información, la especificación de una refactorización compleja –**MOVE METHOD**– junto con una estrategia que permite obtener algunas de las posibles refactorizaciones preparatorias que ayudaría a incrementar las oportunidades de aplicar dicha refactorización. Este enfoque ha sido evaluado mediante dos casos de estudio en los que interviene esta refactorización.

- La corrección de *design smells*, como un tipo especial de proceso complejo de refactorización, puede ser facilitado por medio de la planificación de refactorizaciones.

Esta hipótesis está orientada a un caso más general que el anterior y se dirige a un objetivo más complejo. Para abordarla, se han escrito estrategias de refactorización para dos *design smells*–*Feature Envy* y *Data Class*–. Estas estrategias han sido probadas experimentalmente obteniéndose resultados satisfactorios.

7.3. Resultados relacionados con los objetivos de la tesis

Los objetivos de la tesis formulados en el capítulo 1 se reproducen nuevamente aquí para ser revisados de forma individual:

1. Proporcionar un soporte automático o semi-automático para planificar con antelación las secuencias de refactorizaciones preparatorias –planes de refactorización– que permitan habilitar las precondiciones de un conjunto de refactorizaciones deseado.

Se ha elaborado un enfoque basado en estrategias de refactorización que se instancian en planes de refactorización mediante la planificación de redes de tareas. Este enfoque permite formalizar el conocimiento empírico recabado por los desarrolladores de software y por los usuarios de técnicas de refactorización y de reingeniería, en forma de “recetas”, acerca de cómo se puede aplicar una refactorización determinada, que hemos denominado “estrategias de refactorización”.

Nuestro enfoque permite calcular, a partir de estrategias de refactorización, la secuencia exacta de refactorizaciones que se necesita aplicar para cada caso particular. A dichas secuencias las hemos denominado “planes de refactorización”. El plan de refactorización específico obtenido para cada caso contiene todas las refactorizaciones preparatorias que deben ser ejecutadas para permitir la aplicación de una determinada refactorización deseada. Esto ha sido demostrado mediante la elaboración de una estrategia de refactorización para facilitar la aplicación de **MOVE METHOD**.

El enfoque se ha diseñado de forma que sea completamente automatizable, pero la interacción del usuario puede requerirse para los casos en los cuales el mecanismo de inferencia no es lo suficientemente “inteligente” y durante el proceso de planificación sea necesario solicitar información adicional al usuario.

2. Proporcionar un soporte automático o semi-automático para respaldar la generación de secuencias de refactorizaciones –planes de refactorización–, que permitan transformar un sistema, según una propuesta de rediseño, mientras se preserva el comportamiento del

mismo. Más concretamente, proporcionar un soporte automático o semi-automático para la generación de planes de refactorización orientados a la corrección de *design smells*.

Las estrategias y los planes de refactorización se han definido de tal manera que nuestro enfoque se pueda utilizar para la corrección de *design smells*. El enfoque de planificación de refactorizaciones que hemos desarrollado puede dirigirse al propósito mencionado en el objetivo 1, pero de un modo más general, puede también ser utilizado para obtener el plan de refactorización para cualquier proceso complejo de refactorización que persiga otro objetivo en particular.

En concreto, hemos demostrado que nuestro enfoque puede ser aplicado al problema que motiva esta tesis doctoral: la corrección de *design smells*. Con este propósito, se han escrito y probado estrategias de refactorización para eliminar los *design smells Feature Envy* y *Data Class*.

3. Proporcionar un modo para facilitar a los desarrolladores el uso de las técnicas elaboradas en esta tesis.

Nuestro enfoque de planificación de refactorizaciones se ha diseñado de tal modo que pueda ser usado por tres tipos diferentes de desarrolladores de software.

Los usuarios comunes utilizarán nuestra técnica simplemente para encontrar una manera de aplicar un proceso complejo de refactorización deseado. Seleccionarían la estrategia que quisieran aplicar, completarían los parámetros requeridos, lanzarían la instanciación de la estrategia y esperarían a que el planificador produjera un plan de refactorización. El plan de refactorización podría entonces ser aplicado con la ayuda de una herramienta de refactorización.

Un desarrollador con experiencia suficiente en refactorización y en reingeniería también podría escribir estrategias de refactorización a medida. Esto se ve facilitado por el lenguaje de especificación de estrategias que proponemos en esta tesis y las consultas del estado del sistema que permiten ocultar la complejidad de la representación interna del AST del sistema. Por otra parte, las estrategias de refactorización pueden ser compartidas y reutilizadas. Por lo tanto, estos desarrolladores pueden contribuir a, o beneficiarse de catálogos públicos compartidos de estrategias de refactorización.

Un desarrollador con conocimientos más amplios de nuestro enfoque, relativos a la representación interna empleada, basada en predicados de lógica de primer orden, y especialmente con conocimientos de planificación de redes jerárquicas de tareas, podría escribir consultas del estado del sistema, transformaciones y especificaciones de refactorizaciones adicionales.

4. Evaluar la efectividad, eficiencia y escalabilidad de la propuesta presentada en esta tesis mediante el desarrollo de un prototipo que implemente esta propuesta y la realización de un estudio experimental con el mismo.

Se ha integrado un prototipo de referencia que implementa nuestra propuesta de planificación de refactorizaciones y que sirve para demostrarla. Se ha realizado una evaluación experimental de la propuesta mediante la realización de dos casos de estudio dedicados a la eliminación de los *design smells Feature Envy* y *Data Class* sobre nueve sistemas diferentes con tamaños de pequeños a medios.

7.4. Caracterización de la propuesta desarrollada

A partir de la taxonomía presentada en el capítulo 3, nuestra propuesta se puede describir según las tres dimensiones principales del modelo de características que hemos definido (ver figura 3.1): *design smells*, artefactos y actividades soportadas.

En relación con los **Design Smells** soportados, nuestro enfoque puede ocuparse de cualquier tipo de *design smell* para el que pueda escribirse una estrategia de corrección. Los requisitos que hemos establecido para nuestro enfoque y la representación interna empleada garantizan que cualquier tipo de *design smell* pueda ser gestionado. Sin embargo, el prototipo de referencia presentado en esta tesis solamente aborda dos *design smells*: *Feature Envy* y *Data Class*.

En relación con el **Artefacto** soportado, nuestro enfoque está orientado a código JAVA, utiliza predicados de lógica de primer orden que modelan el AST completo del sistema software como la representación interna del artefacto, y no soporta la gestión de múltiples versiones del sistema. No obstante, nuestra técnica se podría adaptar y modificar para ser utilizada con otros lenguajes de programación y artefactos. De hecho, podría ser empleada con cualquier artefacto para el que se pueda obtener una representación en forma de predicados de lógica de primer orden.

En relación con las **Actividades** soportadas, nuestra técnica abarca: *especificación* de forma manual, para la que el resultado de la actividad es la definición de un dominio de planificación de refactorizaciones; y *corrección* con un nivel de automatización “ejecución según aprobación” para la que el resultado de la actividad es un plan de refactorización. El resto de las actividades de gestión de *design smells* que identificamos en la taxonomía no están soportadas: la actividad de *detección* no se soporta actualmente, pero podría abordarse en el futuro, la actividad de *análisis del impacto* se podría soportar sólo hasta cierto grado y por último, la actividad de *visualización* no se soporta y no prevemos que pueda hacerse en el futuro.

7.5. Comparación con otros trabajos relacionados

Los únicos trabajos similares a nuestra propuesta, relativos a la corrección de *design smells*, de los que tenemos conocimiento, son los desarrollados por Trifu *et al.* [TSG04, Tri08]. Los autores presentan una propuesta para definir y aplicar estrategias de refactorización que, al igual que la nuestra, está basado en “recetas”. De hecho, nuestras estrategias de refactorización se inspiran en el concepto de estrategias de reestructuración de estos autores. Sin embargo, carecen del soporte para la instanciación de las estrategias que nuestra propuesta proporciona mediante planificación automática.

En nuestra propuesta se aplica planificación automática, una técnica de la inteligencia artificial, con el objetivo de resolver un problema de ingeniería de software, consistente en calcular las secuencias de refactorización necesarias para realizar procesos complejos de refactorización. Por lo que sabemos, se trata de un enfoque novedoso, ya que no hemos encontrado ningún trabajo que utilice planificación automática para este propósito en particular. Más aún, sólo hemos encontrado unas pocas referencias relacionadas con la aplicación de planificación automática en ingeniería de software.

Memon *et al.* han utilizado planificación automática en el desarrollo de una herramienta para la generación de casos de prueba de interfaces gráficas de usuario (*GUIs*) [MPS01]. En su trabajo, se analiza la interfaz de usuario para obtener una lista de los eventos permitidos, que se representan como los operadores disponibles en el planificador. El diseñador de las pruebas define las precondiciones y los efectos de los diversos eventos que puede disparar el usuario

y los estados de inicio y meta de los distintos casos de prueba. Los planes producidos por el planificador representan secuencias de prueba que tienen como objetivo la cobertura total de los estados posibles del *GUI*. Los autores emplean un planificador llamado IPP (*Interference Progression Planner*) [KNHD97], que se trata de un planificador de orden parcial perteneciente a la familia de planificadores de tipo GRAPHPLAN.

El planificador elegido parece ser apropiado para el problema que abordan. Sin embargo, sus resultados no se pueden comparar con los nuestros debido a que los dos problemas difieren enormemente en cuanto a su naturaleza y probablemente en cuanto a tamaño. Por otra parte, sus experimentos fueron realizados en 2001 con un ordenador PENTIUM®. Según los autores, su dominio de pruebas de *GUIs* se compone de 32 operadores, y desafortunadamente no especifican el tamaño del estado de sistema. No obstante, podemos especular que no puede acercarse a los tamaños de 32.780 a 354.543 predicados que representan el estado del sistema en nuestro problema. A pesar de que sus resultados no son comparables con los nuestros, sus experimentos también tienen en cuenta técnicas de planificación jerárquica, relacionadas con la planificación de redes jerárquicas de tareas que empleamos en nuestra propuesta. Los experimentos realizados por los autores con y sin técnicas de planificación jerárquica revelan los buenos resultados que ofrecen estos planificadores en términos de eficiencia.

La planificación de redes jerárquicas de tareas (HTN), en concreto el planificador SHOP2, se ha empleado también en la composición de servicios web [SPW⁺04]. Sirin *et al.* abordan el problema de la selección de los servicios web que tienen que ser invocados, de entre una serie de servicios disponibles y en qué orden, con el fin de conformar un servicio web compuesto más complejo. Aunque su problema sea diferente al nuestro, en cuanto a naturaleza y tamaño, algunas de las técnicas que hemos utilizado para materializar nuestro enfoque son similares al suyo. En su trabajo, la definición del dominio HTN se obtiene a partir de las especificaciones de los servicios web disponibles –escritos en la versión de referencia 1.0 del lenguaje de descripción de ontologías para la web (OWL) [DCvH⁺02]– traduciendo estas especificaciones a redes de tareas. Esta traducción se define con reglas sistemáticas. Un enfoque similar se ha seguido en esta tesis para traducir estrategias de refactorización y el lenguaje de especificación de estrategias a elementos de HTN. Incluso, las traducciones de algunos elementos, tales como bucles, se realizan de manera similar a como lo hacen estos autores. Hemos utilizado también variables persistentes con propósitos similares. Estos autores también emplean, como hacemos nosotros, llamadas a procedimientos externos al planificador para solicitar al usuario información adicional durante el proceso de planificación.

Pinna *et al.* han explorado la conveniencia de utilizar planificación de orden parcial para tratar inconsistencias en modelos [PVDSM10]. Sus representaciones del estado del sistema representan modelos UML, en concreto diagramas de clase, y reglas de consistencia. Sus operadores de planificación representan transformaciones simples de modelos y su problema de planificación consiste en la búsqueda de planes orientados a eliminar inconsistencias en modelos, que son representadas como violaciones de las reglas de consistencia. Los autores han experimentado tanto con planificadores parciales que realizan la búsqueda del plan hacia adelante como con planificadores que la realizan hacia atrás, y han concluido que un enfoque basado en estos tipos de planificadores presenta graves problemas de eficiencia incluso para problemas muy pequeños. Esto es debido a que un planificador que realiza la búsqueda del plan sobre el espacio de estados completo y sin ningún conocimiento del dominio que sirva como guía, no resulta eficiente para problemas de gran tamaño. Estos resultados coinciden con las conclusiones que presentamos en el capítulo 5 de la presente tesis.

7.6. Limitaciones de la propuesta desarrollada

La limitación principal del enfoque presentado en esta tesis doctoral está en que se hace necesario escribir el dominio para el planificador HTN –un dominio de planificación de refactorizaciones. Nuestro enfoque no calcula los planes de refactorización explorando el espacio completo de búsqueda de estados, sino que el planificador instancia las heurísticas que se hayan definido en forma de estrategias de refactorización. Estas especificaciones tienen que ser recopiladas a partir de conocimiento empírico y traducidas a redes de tareas. Por consiguiente, la eficacia de la técnica depende en gran medida de la cantidad de estrategias de refactorización, de las especificaciones de refactorizaciones, de otras transformaciones y de las consultas de estado del sistema que se hayan definido e implementado en el dominio de planificación de refactorizaciones. En este trabajo no se ha abordado el problema de cómo recopilar el conocimiento empírico para elaborar las estrategias de refactorización. La escritura de una gran cantidad de estrategias completas está fuera del alcance de esta tesis. En cambio, hemos preferido centrarnos en la definición del enfoque y en construir una implementación de referencia para evaluar y demostrar nuestra propuesta. Como consecuencia de esto, el número de *design smells* y las manifestaciones particulares de ellos que puede manejar nuestro prototipo es algo limitado. Sin embargo, el dominio de planificación de refactorizaciones que hemos elaborado puede servir como referencia y puede ser ampliado y mejorado en el futuro.

Otra desventaja de nuestra propuesta es que la elaboración del dominio de planificación de refactorizaciones es compleja, especialmente si no se cuenta con ninguna herramienta que facilite esta tarea, como ha sido el caso durante el desarrollo de esta tesis. Se trata de una tarea compleja en la que es fácil cometer errores si se realiza de forma manual. Sin embargo, esta limitación puede evitarse mediante la construcción herramientas que faciliten el desarrollo y mantenimiento del dominio de planificación de refactorizaciones. Como un primer paso en esta dirección, se ha propuesto un lenguaje para la especificación de estrategias de refactorización y se ha descrito en el capítulo 5 cómo pueden traducirse los diferentes elementos del dominio de planificación de refactorizaciones a redes de tareas de JSHOP2.

Algunos cálculos son difíciles de implementar en el lenguaje del planificador JSHOP2. Nuestro enfoque sólo puede utilizar la información estructural, léxica y numérica que se puede obtener de AST del sistema.

Es complicado deducir ciertos tipos de información. Por ejemplo, para escribir una estrategia de refactorización para la aplicación de la refactorización **EXTRACT METHOD** el planificador debe ser capaz de averiguar las partes del método que se han de extraer. También pueden presentarse problemas al intentar generar planes que necesiten información semántica, por ejemplo, en el caso de que una estrategia de refactorización necesite identificar si una clase del sistema pertenece a la interfaz gráfica de usuario o a una biblioteca de clases. Estas cuestiones no han sido tratadas en esta tesis, sin embargo, para estos casos nuestro enfoque todavía permite obtener la información necesaria realizando una consulta al usuario. Es más, si la información requerida, que no puede calcularse a través de los mecanismos de inferencia del planificador, puede ser calculada de alguna otra manera por medio de otras herramientas, estas pueden ser invocadas desde el planificador como procedimientos externos.

La herramienta desarrollada durante la elaboración de esta tesis doctoral es un prototipo construido con el propósito de evaluar y demostrar la propuesta presentada. A pesar de que nos ha servido para demostrar la viabilidad de nuestro enfoque, aún se requeriría de bastante trabajo adicional para que esta propuesta pudiera materializarse en una herramienta que fuera realmente

utilizada en un entorno de producción. Como se ha enumerado en los resultados de la evaluación del prototipo en el capítulo 6, nuestra propuesta presenta algunas limitaciones de aplicabilidad debidas a las limitaciones técnicas del prototipo. Por ejemplo, nuestro prototipo no puede ser utilizado con sistemas JAVA que hagan uso de genericidad. Estas limitaciones pueden superarse fácilmente, con el fin de construir una herramienta más integrada, utilizando las versiones más recientes de las herramientas empleadas en el prototipo como JTRANSFORMER y desarrollando una nueva versión del mismo.

Debido a la gran dispersión que presentan los resultados de los experimentos en cuanto a los tiempos de planificación registrados –el tiempo transcurrido durante el cálculo de los planes de refactorización–, no hemos sido capaces de dar una predicción o estimación de cuánto tiempo tardaría nuestro prototipo en obtener un plan de refactorización en función del tamaño del sistema involucrado. Sin embargo, en nuestra opinión, los resultados obtenidos en los sistemas empleados en los experimentos han resultado bastante satisfactorios.

7.7. Resumen de contribuciones

La presente tesis doctoral presenta un enfoque novedoso para la automatización de procesos complejos de refactorización, y en particular para la corrección de *design smells*. También presenta una aplicación novedosa de la planificación automática. No conocemos otras propuestas que empleen planificación automática para este problema en particular –la generación de planes de refactorización para la corrección de *design smells*. Por otra parte, y como ya se ha argumentado en la sección 7.5, por lo que sabemos, sabemos la planificación automática no se ha aplicado apenas en ingeniería de software.

Nuestra propuesta ofrece una infraestructura apropiada para automatizar las estrategias de corrección de *design smells*, porque permite planificar con antelación las secuencias de transformación exactas aplicables para cada caso particular. Las estrategias de refactorización se escriben de una forma homogénea independientemente de si están enfocadas a la aplicación de refactorizaciones complejas, a la corrección de *design smells* o a otros propósitos similares que puedan ser estudiados en el futuro. Debido al uso de un planificador automático, se dispone de algunas estructuras de control no deterministas que permiten escribir especificaciones de procesos complejos de refactorización como estrategias de refactorización. Estas especificaciones pueden ser muy expresivas y pueden emplearse directamente para automatizar reglas heurísticas que de otra forma sólo podrían ser expresadas en lenguaje natural. Nuestra propuesta permite que las estrategias de refactorización puedan escribirse de forma modular y que puedan ser reutilizadas y compartidas.

Las contribuciones de esta tesis se enumeran aquí. Esto puede solaparse en cierta manera con los resultados enumerados previamente en los apartados anteriores, especialmente en la sección 7.1, pero se recogen aquí para hacer hincapié en que se trata de contribuciones al estado del arte. Las contribuciones de esta tesis doctoral se pueden resumir como:

- Una revisión de la literatura relativa a *design smells*, una descripción histórica de los distintos enfoques de gestión de *design smells* y una propuesta terminológica dirigida a clarificar y a unificar los términos y conceptos relacionados con este problema.
- Una revisión exhaustiva del estado del arte de la gestión de *design smells* y una taxonomía basada en modelos de características, todo ello realizado junto con otros autores, para caracterizar los enfoques y las herramientas actuales y futuras.

- La definición del concepto de *estrategias de refactorización* como un modo de escribir especificaciones automatizables de procesos complejos de refactorización.
- La definición de un lenguaje de especificación de estrategias de refactorización para que los desarrolladores de software puedan emplearlo para escribir estrategias para la corrección de *design smells* y otros procesos complejos de refactorización.
- La definición del concepto de *planes de refactorización* como secuencias de refactorización específicas, instanciadas a partir de estrategias de refactorización, que pueden ser aplicadas de manera efectiva sobre un sistema, dado su estado actual.
- La definición de los requisitos que debe cumplir un enfoque para poder ser utilizado para la computación de planes de refactorización.
- Una técnica para instanciar estrategias de refactorización en planes de refactorización por medio de planificación automática.
- Una línea base y un prototipo de referencia para la investigación futura en planificación automática de refactorizaciones.

7.8. Trabajo futuro y preguntas abiertas

Nuestra primera prioridad como trabajo futuro está dirigida a superar las limitaciones actuales de nuestra propuesta. En concreto, esperamos mejorar el dominio de planificación de refactorizaciones y desarrollar una herramienta que sea realmente usable a partir del prototipo de la referencia construido durante la elaboración de esta tesis.

Para mejorar el dominio de planificación de refactorizaciones, en primer lugar tiene que ser adaptado para utilizar la última versión del formato de representación de ASTs basado en lógica de primer orden de JTRANSFORMER. Más adelante, para aumentar la cobertura del dominio de planificación de refactorizaciones, este deberá ser aumentado añadiendo más estrategias de refactorización para la corrección de *design smells*, las especificaciones de más refactorizaciones y más estrategias de aplicación de refactorizaciones complejas. El dominio de planificación de refactorizaciones también tiene que mejorarse añadiendo más consultas, especialmente para el cálculo de métricas. Incorporar este tipo de operaciones permitiría ejecutar también como consultas en nuestro planificador las estrategias de detección de *design smells* de Lanza y Marinescu [LM06]. Como consecuencia de esto, se podría mejorar la calidad de los planes producidos, al permitir que el planificador pudiera seleccionar las transformaciones que se fueran a incluir en el plan de forma más precisa, ya que podrían identificarse de una forma más apropiada las entidades involucradas en el *design smell*.

También debemos investigar en el futuro cómo podrían realizarse o incluirse algunas consultas más complejas. Por ejemplo, como ya se ha mencionado, debería estudiarse cómo podría obtenerse determinada información semántica, cómo podrían deducir las porciones de un método que deberían extraerse al aplicar la refactorización **EXTRACT METHOD**, cómo podría distinguirse una clase de la interfaz gráfica de una clase de la biblioteca, etc.

Como se ha adelantado también en las conclusiones del estudio de casos (capítulo 6), pretendemos desarrollar una nueva versión del prototipo que pueda ofrecerse como una herramienta realmente usable. Se ha desarrollado un prototipo sencillo utilizando tecnologías existentes y

otros prototipos. En el futuro se debería abordar el desarrollo de una versión mejorada de esta herramienta. Ésta debería estar integrada con la versión más reciente de JTRANSFORMER y esto implicaría traducir el algoritmo de planificación de JSHOP2 a PROLOG, y construir la herramienta como un *plugin* de ECLIPSE.

También podría explorarse en un futuro el uso de otros formatos para la representación interna de los artefactos con el fin de mejorar la aplicabilidad de nuestra propuesta. Por ejemplo se podría estudiar la posibilidad de emplear un modelo de representación de software orientado a objetos como MOON [Cre00, LMCP06, MLCP07]. Este modelo proporciona soporte para genericidad e independencia del lenguaje y por lo tanto puede mejorar la aplicabilidad de nuestra técnica.

También consideramos que es necesario desarrollar herramientas para facilitar la elaboración del dominio de planificación de refactorizaciones como redes HTN. Se debería desarrollar un compilador para el lenguaje de especificación de estrategias de refactorización. También nos gustaría ofrecer una herramienta para poder escribir y depurar estrategias de refactorización y especificaciones de refactorizaciones de forma visual, con la ayuda de un editor gráfico. Esta herramienta podría estar basada en transformaciones de modelos, de forma que las especificaciones de refactorizaciones desarrolladas de forma gráfica puedan ser transformadas automáticamente en redes de tareas siguiendo las reglas elaboradas en el capítulo 5.

Sería útil también disponer de una herramienta para ejecutar baterías de experimentos que permita obtener fácilmente los análisis de los resultados obtenidos. Esta herramienta para pruebas se emplearía en el desarrollo del dominio de planificación de refactorizaciones y también podría ayudar a evaluar este dominio con más detalle. Por ejemplo, será interesante para poder determinar la cantidad de conocimiento más apropiada para obtener los mejores resultados en términos de efectividad, eficiencia y escalabilidad.

Nuestro prototipo solamente devuelve el primer plan encontrado. Otra posible manera de mejorar la herramienta, sería explorar la posibilidad de generar múltiples planes. Sería útil poder obtener múltiples planes, cuantificar sus diferencias en términos de factores de calidad y ofrecerle al usuario el mejor plan, o informarle sobre los diferentes planes alternativos y su respectiva calidad.

Otra línea de trabajo futuro podría dedicarse a explorar otras posibles aplicaciones de nuestro enfoque, diferentes de las que motivaron nuestro estudio y que se han presentado en esta tesis. Como ejemplo, sería interesante escribir estrategias de refactorización para otros objetivos tales como la introducción de patrones de diseño. Finalmente, nos parece interesante explorar cómo podrían ampliarse las capacidades de nuestra propuesta, explorando técnicas alternativas para soportarlo o empleándolo en situaciones diferentes que no se hayan contemplado.

Otro posible uso de nuestra propuesta podría ser la investigación y el desarrollo de refactorizaciones para entornos integrados de desarrollo y herramientas de refactorización. Las estrategias de refactorización, desarrolladas para nuestra técnica, que estuvieran lo suficientemente maduras podrían ser trasladadas a una herramienta de refactorización como refactorizaciones de alto nivel. Nuestro enfoque se podría utilizar para desarrollar de forma incremental una estrategia de refactorización, hasta que se alcanzara un nivel de detalle en el que los mecanismos de inferencia y no-determinismo del planificador ya no fueran necesarios. Gracias al planificador de refactorizaciones, podría obtenerse una estrategia de refactorización con un nivel de detalle que permitiera implementarla como una refactorización disponible dentro de un entorno de desarrollo integrado u otra clase de herramienta de refactorización.

PART II

**REFACTORING PLANNING FOR DESIGN SMELL CORRECTION IN
OBJECT-ORIENTED SOFTWARE**

(ENGLISH)

Contents

1	Introduction	1
1.1	Overview of the problem	3
1.1.1	What is the problem to solve?	3
1.1.2	Why is it a problem?	3
1.1.3	Why is it an important problem?	4
1.2	Thesis statement, objectives and contributions	7
1.2.1	Objectives	7
1.2.2	Summary of contributions	7
1.3	Structure of the dissertation	8
2	Context	11
2.1	Design smells: definitions and terminology	11
2.1.1	On the notion of the term “ <i>smell</i> ”	12
2.1.2	Code smells	12
2.1.3	Design smells	13
2.1.4	Historical background on design smells	15
2.2	Refactoring	17
2.2.1	Applying refactorings	18
2.2.2	Developing refactoring tools	19
2.2.3	Mining refactorings	19
2.2.4	Suggesting refactorings	20
2.3	Design smell correction with refactorings	21
3	A Survey on Software Design Smell Management	23
3.1	Overview of the survey	23
3.1.1	Feature modelling notation	24
3.1.2	Top level features of the design smell management survey	25
3.2	Design smell	26
3.3	Target artefact	28
3.3.1	Type of artefact	28
3.3.2	Versions	29
3.3.3	Type of representation	29
3.4	Activity	30
3.4.1	Specification	31
3.4.2	Detection	32
3.4.3	Visualisation	34

3.4.4	Correction	34
3.4.5	Impact analysis	35
3.5	Conclusions of the survey	37
4	Refactoring Strategies	39
4.1	Analysis of design smell correction specifications	39
4.1.1	Current design smell correction specifications	39
4.1.2	A model for current design smell correction specifications	47
4.1.3	Current specifications of refactorings	50
4.1.4	The activities of applying a design smell correction strategy	52
4.2	Open problems in automating correction strategies	53
4.2.1	Applicability of refactorings	53
4.2.2	Non-formal description of correction strategies	54
4.3	Definition of refactoring strategies	55
4.3.1	Overview of refactoring strategies	55
4.3.2	A model for refactorings and non-behaviour-preserving transformations . .	57
4.3.3	A model for refactoring strategies	59
4.4	A small language to specify refactoring strategies	63
4.5	Characteristics of the problem	69
5	Refactoring Planning	73
5.1	Lessons learned from previous works	73
5.2	A brief introduction to automated planning	77
5.2.1	Formal basics of automated planning	79
5.2.2	The restricted model of classical planning	81
5.2.3	Characterisation of the refactoring planning problem	82
5.2.4	A variety of planners	85
5.3	JSHOP2: a hierarchical task network planner	89
5.3.1	HTN planning	89
5.3.2	Features of JSHOP2	91
5.3.3	Elements of a JSHOP2 planning problem	92
5.3.4	Managing variables and variable scopes	99
5.4	Refactoring strategies as HTNs	102
5.4.1	System elements	103
5.4.2	System queries	103
5.4.3	Non-behaviour-preserving transformation steps	104
5.4.4	Non-behaviour-preserving transformations	112
5.4.5	Refactorings	113
5.4.6	Invocations and queries of refactoring strategies	113
5.4.7	Non-deterministic alternatives	114
5.4.8	Non-deterministic loops	115
5.4.9	Unordered strategy steps	117
5.4.10	Refactoring strategies specification language as JSHOP2 HTNs	119
5.5	Computing a refactoring planning problem	119
5.6	How JSHOP2 meets the requirements of refactoring planning	121
5.6.1	General requirements	121
5.6.2	Requirements as a planning problem	122

6	Case Study	125
6.1	A refactoring planning domain	125
6.1.1	Refactorings	125
6.1.2	Refactoring strategies	131
6.2	A refactoring planner prototype	146
6.2.1	Tools used in our prototype	146
6.2.2	How the tools are integrated into our prototype	149
6.3	Description of the case study	151
6.3.1	Experiments setup	151
6.4	Analysis of the results	155
6.4.1	Discussion on the produced plans	155
6.4.2	Discussion on the prototype efficiency	159
6.5	Conclusions of the case study	172
6.6	Characterisation of the refactoring planning approach	174
6.6.1	Regarding design smell	174
6.6.2	Regarding target artefact	175
6.6.3	Regarding activities	175
7	Conclusions	177
7.1	General results	177
7.2	Results regarding thesis statements	178
7.3	Results regarding thesis objectives	180
7.4	Characterisation of the approach	181
7.5	Comparison with related works	181
7.6	Limitations of the approach	183
7.7	Summary of contributions	184
7.8	Future work and open questions	185
	References	186
A	JSHOP2 Planning Language Specification	205
A.1	Symbols	205
A.2	Terms	205
A.3	Logical atoms and logical expressions	206
A.4	Logical preconditions	207
A.5	Axioms	207
A.6	Task atoms and task lists	207
A.7	Operators	208
A.8	Methods	208
A.9	Planning domains	208
A.10	Planning problems	209
B	Translation of the refactoring strategy language	211
B.1	Strategy Definitions	211
B.2	Sequences of steps	212
B.3	Managing variables between methods	214
B.4	Managing lists	215

B.5	Queries, invocations and boolean expressions	215
B.6	Variables and literals	216
B.7	Alternatives	217
B.8	Loops: while	218
B.9	Loops: foreach	219
B.10	Loops: foreach	221
B.11	Unordered steps	223
C	The Refactoring Planning Domain	225
C.1	Notation	225
C.2	Refactoring Strategies	225
C.3	Refactorings	226
C.4	Operators	226
C.5	External Java Procedures	227
C.6	System Queries	227

List of Figures

2.1	A brief history of design smell management.	16
3.1	Feature diagram notation used throughout this survey.	24
3.2	Top-level variation points for comparing design smell management approaches. .	25
3.3	Top-level <i>Design Smell</i> feature, and its subfeatures.	26
3.4	Top level <i>Target Artefact</i> feature, and its subfeatures.	28
3.5	Top level <i>Activity</i> feature, and its subfeatures.	31
3.6	Degree of automation support of a design smell management activity.	31
3.7	Summary of the current situation for the specification activity.	32
3.8	Summary of the current situation for the detection activity.	33
3.9	Summary of the current situation for the visualisation activity.	35
3.10	Summary of the current situation for the correction activity.	36
3.11	Summary of the current situation for the impact analysis activity.	37
4.1	A general model for current design smell correction strategies.	48
4.2	A simplified model for refactoring operations.	51
4.3	Refactoring strategies and their relationship with refactorings and non-behaviour-preserving transformations.	56
4.4	A model for the refactorings and the non-behaviour-preserving transformations used in refactoring strategies.	58
4.5	A model for refactoring strategies.	60
4.6	Overview of the relations between strategies, refactorings and transformations. . .	64
5.1	Complete planning state graph for the apples and book problem.	79
5.2	Conceptual models for dynamic planning and the refactoring planning problem. .	80
5.3	Elements of a hierarchical task network used in HTN planning.	89
5.4	A simplified model of JSHOP2's HTNs	92
5.5	A model for HTN planning problems	98
6.1	Root of the MOVE METHOD refactoring design.	126
6.2	Design of the move-method-transformation task.	127
6.3	Design of the MOVE METHOD refactoring variation	128
6.4	Design of a transformation for updating references	129
6.5	Design of a transformation for adding “this” references	130
6.6	Design of a transformation for replacing “this” references.	131
6.7	Overview of a simple strategy for applying the MOVE METHOD refactoring. . .	137
6.8	Overview of a simple strategy for removing a <i>Feature Envy</i> smell.	140

6.9	Overview of simple strategies for removing a <i>Data Class</i>	143
6.10	Overview of the tools used in the experiments and how they are integrated. . . .	150
6.11	Overview of the experiment’s execution environment.	154
6.12	Distribution of the strategy “paths” traversed for computing the plans obtained. .	156
6.13	Number of refactorings and transformations in the produced plans.	158
6.14	Distribution of the elapsed time for the <i>Feature Envy</i> experiments per system . .	162
6.15	Distribution of the elapsed time for the <i>Data Class</i> experiments per system. . . .	163
6.16	Distribution of the planning time for the two case studies and per system.	165
6.17	Probabilistic upper bounds for planning time per factbase size.	171

List of Tables

5.1	Interpretation of the refactoring planning problem as a graph grammar problem.	74
5.2	Interpretation of the refactoring planning problem as a state-space search problem.	75
5.3	Classical Planning Operators (STRIPS)	77
5.4	Specification of the apples and book problem.	78
5.5	Interpretation of the refactoring planning problem as an HTN planning problem.	91
6.1	Details of the system used in our experiments.	153
6.2	Summary of the results.	155
6.3	Summary of the strategies traversed.	156
6.4	Summary of the plans produced.	158
6.5	Correlation between system size and mean elapsed time.	160
6.6	Descriptive statistics of the results.	164
6.7	Quartiles and ranges of planning time.	164
6.8	Results of the normality tests for the planning time variable.	167
6.9	Results of the Kruskal-Wallis tests for the planning time variable.	168
6.10	Results of the Mann-Whitney (Wilcoxon) test for planning time.	169
6.11	Probabilistic upper bounds for planning time.	171

Chapter 1

Introduction

Software evolution is a fundamental part of the software development process, that often results in an increase of software entropy and, as a consequence, in the decay of software structure [EGK⁺01]. This mainly happens because maintenance efforts concentrate more on bug correction and the addition of new functionality, than on the control and improvement of the system's architecture and design [Bro75].

Problems in the software's structure can manifest in the form of design smells. Design smells [BF99a] are design problems that come from “poor” design choices, leading to ill-structured software. This may hinder further development and maintenance¹ by making it harder for software developers to change and evolve the software. Design smells do not produce compile or run-time errors, but negatively affect software quality factors. In fact, this negative effect on quality factors could lead to real compile and run-time errors in the future. Correcting or, at least, reducing design smells can improve software quality. The most desirable approach would be to prevent them, but a systematic technique to detect and correct software design smells once they have appeared is still needed. Since the aim of design smell correction is not to change the system's behaviour –the system is working, it is not necessary to fix its behaviour– design smells are corrected with refactorings.

Refactorings are source code transformations that change the system's internal design while preserving its observable behaviour [Opd92]. Refactorings can be used to improve, in general, certain quality factors such as reusability, understandability, maintainability, etc. More specifically, refactorings can help to achieve a particular system structure, such as introducing a design pattern [Ker04] or consolidating the system's architecture [NL06].

Refactoring operations are defined in terms of preconditions and transformations. Preconditions define the situations under which a refactoring operation can be safely applied without altering the system's behaviour. The transformation part specifies which are the changes to be performed to the system. Refactoring operations are meant to be executed in small steps, so that more complex refactorings can be executed by the composition of simpler ones. Behaviour preservation is also easier to check in the case of simpler refactorings.

When a refactoring process aims to solve a complex problem, such as the correction of design smells, a significant amount of changes is needed. Refactorings' preconditions can help to assure behaviour preservation, but at the same time they hinder the application of complex transformation sequences because they restrict the applicability of refactoring operations. If any

¹The maintainability is the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment [ISO01].

precondition of any operation in the intended transformation sequence, is not fulfilled at the time of its application, the whole sequence can not be applied. This makes it hard for the developer to perform complex refactoring processes, such as the correction of design smells.

Some authors have tried to circumvent this problem [KK04]. Static composition of refactorings avoids the violation of the preconditions of the intermediate refactorings for predefined or fixed refactoring sequences. The definition of the refactorings to be applied, in the case of design smell correction, is made in terms of correction strategies which are mainly heuristics. The application of this kind of refactoring sequences has an increased difficulty since the intended correction strategy has to be instantiated for each specific use. This means that the developer has to find out the suitable refactoring sequence for each particular case from the wide range of choices defined by the correction strategy.

Several techniques have been developed to support the different activities of design smell management. For example, metrics, structure analysis, and other techniques have been proved to be very useful in revealing smells and in suggesting changes to reduce or to remove them. A convenient way to present these suggestions is through the proposal of refactorings. The problem is that, as already mentioned, it is rare that the preconditions of the desired transformations could be fulfilled by the system's source code at its current state. This means that additional transformations are needed in order to "prepare" the system for the desired evolution. Therefore, suggesting refactorings is not enough to allow systematic correction of design smells. For the correction activities, reviewing the state-of-the-art reveals there is a need for an approach to plan and generate executable design correction strategies. The suggestions, that the existing approaches produce for correction, cannot be executed as they are.

This PhD thesis dissertation tackles this problem. This research is aimed at improving the automation of the refactoring activity when it is oriented to the correction of design smells. Indeed, any activity involving complex refactoring processes might be improved if we can provide a more automated method to compute the sequence of refactoring operations needed to accomplish a certain objective. The approach proposed is based on the generation of refactoring plans. We define refactoring plans as the specification of refactoring sequences, that match a system redesign proposal or a correction suggestion, and which could be generated to apply a smell correction specification in each particular case. Thereby the desired refactorings can be executed over the current system's source code. Refactoring plans allow to compute the preparatory refactorings needed, thus helping the developer to circumvent the violation of refactoring preconditions.

Formal theories, such as graph transformation or first order logic, can be used to analyse a set of available refactorings, within a context given by the current system's source code and a redesign proposal, in order to obtain the refactoring plan. We ourselves have explored the graph transformation approach before developing the proposal presented in this Thesis. This exploration has served us for gaining insight and acknowledgment in the problem. This Thesis analyses the current state of the art on correction of design smells. This helps us to define the main characteristics of the problem and to model a solution by introducing the concept of **Refactoring Plans**. The approach presented in this dissertation to improve the automation of refactorings is based on automated planning [GNT04], and more specifically on hierarchical task network planning [GNT04, chapter 11]. This technique is analysed to reveal its suitability for the refactoring planning problem. We show finally, how refactoring plans can be supported with hierarchical task network planning and demonstrate its application with a case study.

1.1 Overview of the problem

1.1.1 What is the problem to solve?

The problem to solve in this PhD Thesis Dissertation is to give an automated or semi-automated support to refactoring planning in the use of refactorings for correcting Object-Oriented software design smells. This will involve the definition of a framework for the instantiation of design-smell-removing suggestions into correction plans which could be effectively applied through refactorings, because the system's behaviour should be preserved. This problem can be encompassed within a wider scope. The results derived from this dissertation could benefit other scenarios where complex refactoring sequences are needed in order to achieve a particular design goal or improvement.

The main problem to solve will be to give automated or semi-automated support to refactoring planning in the context of using refactorings to correct software design smells.

1.1.2 Why is it a problem?

Techniques to detect design smells and correct them are maturing and increasing in number: detection of bad smells with metrics [LM06, CLMM06], usage of structural patterns to find smells [Moh08], Formal/Relational Concept Analysis to propose reorganisation of Object-Oriented entities [PCML03, MBG06], just to cite a few of them. The open problem is still that all the change suggestions given by the existing approaches and tools are not directly applicable over a system. They are given in terms of refactorings, but these refactorings are rarely directly applicable. Additional and preparatory refactorings and complex refactoring sequences have to be planned ahead. It does not matter how precisely these refactoring suggestions are given out [TSG04]. They can only be general specifications that have to be instantiated in a very specific situation, for a particular system at a particular state. Those design smell correction descriptions can not anticipate every possible instantiation. Therefore, the responsibility to instantiate this smell correction specifications is entirely left to the developer. The refactorings to execute are often very complex sequences, quite different from the ones originally suggested by the smell management tool.

Not only refactorings have to be planned ahead when they are applied in the particular scenario of correcting design smells, but also when they are employed for any other objective on a daily basis. According to the study conducted by Murphy-Hill and Black in [MHB08], there is a need to improve the usability of refactoring tools in order to achieve a wider adoption of this technique by developers. Specifically, they found that precondition violation is a problem which has not been addressed yet. When a developer tries unsuccessfully to apply a refactoring with a refactoring tool, most of the errors identified are violations of the refactoring preconditions. Other problems included how to understand and avoid these violations and how to decide which refactorings to apply. At the current state of refactoring tools, the assistance offered by the IDEs proves not helpful enough. Current tools just print error messages with few useful details. These authors propose to improve refactoring tools with better user interfaces that could help the developer to understand the precondition violations.

Thus, there is a clear need to improve the automated assistance for the developer when it comes to understand precondition violations and how to solve them in order to apply a desired refactoring.

1.1.3 Why is it an important problem?

Refactoring involves “many refactorings”

Developing a solution for refactoring precondition fulfillment opens the door to an improvement on the automation of complex refactoring processes, such as design smell correction with refactorings. The automated generation of complex refactoring sequences, aimed at enabling refactoring preconditions, can help to apply a desired single refactoring or a set of refactorings. The approach can then be easily extended and generalised for planning refactoring sequences aimed at, for example, design smell correction.

The problem of precondition fulfillment can help improve the refactoring process in a wide variety of scenarios. In a refactoring process, refactoring operations are not executed in an isolated way. Instead, refactorings are applied in sequences, often encompassing wider changes to the code. Some examples of scenarios where complex refactoring sequences appear are:

- Refactoring compositions, defined by Opdyke, such as “high-level refactorings” [Opd92, page 79], and “composite refactorings” [Opd92, page 74]. Complex refactorings like these can be built upon simpler or lower-level ones. For example, to apply the high-level refactoring **CREATING AN ABSTRACT SUPERCLASS** [Opd92, page 79], Opdyke compiles a specification that includes low-level refactorings such as: **CREATE EMPTY CLASS** [Opd92, page 56], **CREATE MEMBER FUNCTION** [Opd92, page 57], **DELETE MEMBER FUNCTIONS** [Opd92, page 59], **CHANGE VARIABLE NAME** [Opd92, page 60], **CHANGE MEMBER FUNCTION NAME** [Opd92, page 61], **CHANGE TYPE** [Opd92, page 62], **CHANGE ACCESS CONTROL MODE** [Opd92, page 63], **REORDER FUNCTION ARGUMENTS** [Opd92, page 66], **ADD FUNCTION BODY** [Opd92, page 67], **MOVE MEMBER VARIABLE TO SUPERCLASS** [Opd92, page 72], and composite refactorings such as: **CONVERT CODE SEGMENT TO FUNCTION** [Opd92, page 75], **MOVE CLASS** [Opd92, page 76]. Composite refactorings are refactorings built upon low-level ones but involving less complexity than high-level ones.
- Refactoring specifications from Fowler *et al.* [FBB⁺99]. Refactorings can enable or disable the precondition of other refactorings. Fowler *et al.* describe these dependencies among refactorings in his specifications. In order to accomplish the execution of a particular refactoring operation, this process must often be performed along with the execution of other refactorings which enable the precondition of the first one. For example, **REPLACE CONDITIONAL WITH POLYMORPHISM** [FBB⁺99, page 255] often requires executing **REPLACE TYPE CODE WITH SUBCLASSES** [FBB⁺99, page 223], **EXTRACT METHOD** [FBB⁺99, page 110], **MOVE METHOD** [FBB⁺99, page 142], etc.
- Refactorings aimed at a certain goal, such as introducing or removing design patterns [Ker04], or removing design smells [BF99a]. As an example, according to Fowler *et al.*, in order to remove a *Data Class*, one should have to apply a number of refactorings including; **ENCAPSULATE FIELD** [FBB⁺99, page 206], **ENCAPSULATE COLLECTION** [FBB⁺99, page 208], **REMOVE SETTING METHOD** [FBB⁺99, page 300], **MOVE METHOD** [FBB⁺99, page 142], **EXTRACT METHOD** [FBB⁺99, page 110] and **HIDE METHOD** [FBB⁺99, page 303].

All these scenarios can benefit from an approach which enables the automated generation of refactoring sequences or plans, to build up complex refactorings. This PhD thesis dissertation is driven by the last case, but the first two cases are also indirectly supported.

Regarding design smells

As stated by Lehman in his *laws of software evolution* [Leh96], it is a well known and verified fact that software evolves or else, it becomes obsolete and progressively less useful:

“An E-type program² that is used must be continually adapted else it becomes progressively less satisfactory.” **I - Law of Continuing Change**

Moreover, as already mentioned at the beginning of this chapter, software design decay is a recurrent process [Bro75, EGK⁺01]. One way to address design decay is through the detection and correction of design smells, *i.e.*, what Kerievsky calls “*paying design debt*” [Ker04] or Neil *et al.* call “*strategic refactoring*” [NL06].

Design smells can certainly have a bad influence on software quality factors. Although, up to our knowledge, there are not any really conclusive experiments on the subject, the common software engineering knowledge confirms this idea.

In his PhD dissertation [Mar02], Marinescu uses quality models to establish the relationships between design smells and software quality factors. He calls these models *factor strategy models* [Mar02, pages 87–108]. This mapping between design smells and software quality factors allows to describe which design smells can have a negative impact over a certain quality factor, and vice versa, which quality factors can be deteriorated by a particular design smell. The relationships defined within these models are weighted, so that the degree of impact of each design smell can be taken into account too. He also applies this approach to present a model definition for maintainability [Mar02, pages 145–147]. With this maintainability model he tries to describe how this quality factor can be negatively affected by some design smells, such as *Feature Envy*, *Temporary Field*, *Shotgun Surgery*, *Refused Bequest*, *God Class*, *God Package*, *God Method*, *Data Class*, etc.

Marinescu also presents a case study to evaluate the factor strategy models [Mar02, pages 109–130]. In his PhD dissertation, two versions of a large-sized business application related to computer-aided route planning were analysed. The first version is composed of 93,000 lines of code, 18 packages, 152 classes and 1284 methods. The second version, containing 115,600 lines of code, 29 packages, 387 classes and 3446 methods, has grown in functionality while at the same time it has been redesigned to remove some of the design smells detected within the first version. The author states that the use of factor strategy models shows how maintainability is significantly improved between the two versions of the system.

In [LWN07a], Lozano *et al.* argue that in order to evaluate the impact of design smells over the maintainability of a system, the evolution of the system and design smells themselves have to be analysed through the different versions of the system. On the basis of this premise, the same authors present in [LWN07b] a case study for the design smell known as *Duplicated Code* (they actually use the term “code clones”). This preliminary experimental work revealed that clones introduce implicit dependencies in the code, causing an increased maintenance effort and the appearance of other design smells like shotgun surgery.

Ratzinger *et al.*, describe in [RFG05] how they conducted a succesful case study for detecting and correcting design smells in an industrial picture archiving and communication system. In

²E-type refers to the software that models human processes, organisation and activities. This type of software naturally evolves as its problem domain does. The “E” stands for evolutionary. Other software types defined by Lehman are: S-Software, which can be formally derived from the specification; and P-Software, which can be specified and derived formally but the actual computation has to be addressed through heuristics and approximation [Leh80].

this paper, they explain how correction of change-coupling related smells lead to an improvement on the evolvability of the system.

Certain authors try to explore in which situations a bad smell reveals as bad design and in which cases it does not. Even though duplicated code is considered a design smell that affects maintainability in a negative way –Fowler *et al.* mention it as “Number one in the stink parade” [FBB⁺99, page 76]–, there are some specific situations in which duplicated code can be considered a good design choice. In [KG06], Kasper *et al.* try to compile eight patterns of duplicate code usage (they use the term “code clones”) to analyse in which cases it is a problem. They also describe the special cases in which duplicated code is not a problem but, on the contrary, it can be the best design choice. In the end, the paper is a warning to smell detection practices. It also confirms the meaning of the term “smell”: not all smells are actually design problems. An automated procedure to manage design smells has to provide a way to discern between these cases, either automatically or in a developer-assisted way.

The improvement produced through refactoring, not necessarily aimed at the correction of design smells over the design of a system and, as a consequence, over its quality factors, has been evaluated and logged by many other authors.

Kataoka *et al.* describe in [KIAF02] a method to evaluate the effect of refactoring over the maintainability of the system by computing coupling metrics. The case study presented in this paper shows how the application of certain refactorings actually enhanced the maintainability of the system.

Moser *et al.* present in [MAP⁺08] a case study aimed at assessing the impact of refactoring in a close-to-industrial environment. The system analysed is a commercial software product to monitor applications for mobile Java-enabled devices. The software project was carried out in an Agile development environment. The development team was composed of professional developers and students. Their results indicate that refactoring not only increases aspects of software quality, but also improves productivity.

Strogylos *et al.* analyse in [SS07b] whether refactoring is being effectively used as a means to improve software quality within the open source community. In order to do this, they evaluate the logs of the version-control system hosting the source code of some popular open source software systems, such as APACHE HTTPD, APACHE LOG4J, MYSQL CONNECTOR/J and JBOSS HIBERNATE, to detect changes marked as refactorings. They examined how the software metrics are affected by the usage of refactoring practices. They conclude that not every refactoring process produces design and quality improvement. On the contrary, their findings showed how some metrics, such as LCOM –lack of cohesion of methods [CK91]– can increase, indicating a worse design, if developers do not apply refactorings in an effective way.

Neil *et al.* argue in [NL06] that refactoring has to be planned ahead in order to be successful, always having the objective of design improvement in mind. They suggest to follow a “*strategic refactoring*” approach which they describe as “design-pattern-based refactorings with architectural awareness”. This technique advises to plan refactorings through the correction of design defects and the introduction of design patterns. They indicate that according to their studies, strategic refactoring using design patterns is the most effective way to repair decaying code for Object-Oriented systems. They review a case study where this has been successfully applied, but they warn that design can deteriorate further if refactorings are misapplied.

1.2 Thesis statement, objectives and contributions

This dissertation is structured around this thesis statement:

The activity of refactoring, when complex refactoring sequences have to be applied, as in the case of design smell correction in Object-Oriented software, can be assisted by means of refactoring plans that can be obtained automatically.

The thesis statement is addressed by decomposing it into the following research hypotheses:

- State of the art in automated design smell management is mature in detection but still has to be improved in correction.
- Refactoring Suggestions produced by current design smell detection tools are not directly applicable.
- Design smell correction with refactorings corresponds to the general schema of applying complex refactoring sequences with a strategic objective.
- Complex refactorings can be assisted with refactoring planning, by enabling refactoring preconditions with preparatory refactorings that can be obtained automatically.
- Design smell correction, as a special kind of complex refactoring process, can be assisted by means of refactoring planning.

1.2.1 Objectives

The dissertation is aimed at the following objectives:

1. Provide an automated or semi-automated support to plan ahead the preparatory refactoring sequences –refactoring plans– that can enable the precondition of a desired set of refactorings.
2. Provide an automated or semi-automated support to assist the generation of refactoring sequences –refactoring plans– that can transform a system, following a redesign proposal while preserving the system’s behaviour. More specifically, to provide an automated or semi-automated support to the generation of refactoring plans for design smell correction.
3. Provide a way to help software developers use the techniques elaborated in this PhD Thesis Dissertation.
4. Evaluate the effectiveness, efficiency and scalability of the approach presented in this PhD Thesis Dissertation by developing a prototype which implements this approach and by performing an experimental study with it.

1.2.2 Summary of contributions

This PhD thesis dissertation presents the following contributions:

- A review on the design smells’ literature, a historic overview of design smell management approaches, and a terminology proposal, aimed at clarify and unify the terms and concepts related to this subject.

- A survey on design smell management and a taxonomy, based on feature models and co-written with other authors, to characterise the present and future approaches and tools.
- The definition of refactoring strategies as a means of writing automation-suitable specifications of complex refactoring processes.
- The definition of a refactoring strategy specification language that software developers can use to write strategies for design smell correction and other complex refactoring processes.
- The definition of refactoring plans as specific refactoring sequences, instantiated from refactoring strategies, that can be effectively applied over a system in its current state.
- The definition of the requirements an approach has to fulfill in order to support the computation of refactoring plans.
- A technique to instantiate refactoring strategies into refactoring plans by means of automated planning.
- A base line and reference prototype for future research in automated refactoring planning.

1.3 Structure of the dissertation

The reminder of this dissertation is structured as follows.

Chapter 2 presents the context of the problem that motivates this dissertation. It describes what design smells are, while proposing an unifying terminology for this kind of software problems. A historical overview on design smells is also performed. The chapter also comprises a brief introduction to refactoring in general and refactoring automation in particular and describes the relationship between design smells and refactorings. This reveals that the need to improve the design smell correction activity can drive the improvement of the automation of complex refactoring processes.

Chapter 3 reviews the state of the art in design smell management. It compiles a comprehensive survey on the approaches and tools that relate to the detection, correction, visualisation, etc. of design smells. The analysis is performed through the definition of a taxonomy, based on feature models, that serves to identify, characterise and compare the existing approaches and the future ones.

Chapter 4 defines refactoring strategies and refactoring plans. First of all, this chapter analyses how design smell correction is currently supported through heuristic correction specifications. Refactoring strategies and refactoring plans are presented then, as a means for writing more formal specifications for complex refactoring processes, that can foster the automation of the design smell correction activity. Finally, this chapter describes the characteristics of the problem of automating the instantiation of refactoring strategies into refactoring plans.

Chapter 5 discusses which are the most convenient techniques to support the automated instantiation of refactoring strategies, and presents hierarchical task network planning as the technique selected. A brief introduction on automated planning and on hierarchical task network planning is included. The remainder of this chapter is dedicated to an analysis on how refactoring strategies can be translated and automated on top of this technique.

Chapter 6 describes a case study that validates the approach presented in this dissertation. The specifications for correcting two design smells –*Feature Envy* and *Data Class*– are compiled

into refactoring strategies and translated then into a refactoring planning domain, in order to be automated. The refactoring planner is executed over a set of open source systems, which present several manifestations of this design smell, to obtain the proper refactoring plans that will remove the smell. This chapter describe these experiments and discuss their results. The design smell correction proposal presented in this dissertation is also characterised using the taxonomy defined in Chapter 3.

Chapter 7 summarises the outcome of this dissertation, discusses its results, contributions and limitations and presents the open research perspectives.

Chapter 2

Context

2.1 Design smells: definitions and terminology

Bad design practices, often due to inexperience, insufficient knowledge or time pressure, are at the origin of design smells. They can arise at different levels of granularity, ranging from high-level design problems, such as antipatterns [BMMIM98], to low-level or local problems, such as code smells [BF99a]. Often, these problems are not isolated, but are usually symptoms of more global defects. Design smells are quite different from software *defects* (also referred to as *bugs*) which are “*deviations from specifications or expectations which might lead to failures in operation*” [FN99, Hal77].

A number of techniques and tools have been proposed in the literature both for the *detection* and *correction* of design smells. The *detection* techniques proposed consist mainly in defining and applying rules for identifying design smells. The *correction* techniques often consist in suggesting which refactorings could be applied to the source code of a system to restructure it, thereby correcting, or at least reducing, its code and design problems.

Design smell management refers to the techniques, tools and approaches addressed to detect, correct or reduce design smells. Correcting, or at least, reducing smells can improve software quality. As such, the goal of a smell management process is to change the system’s structure to improve its internal quality factors, in particular its understandability and maintainability¹. The activities of design smell management range across: smell specification, information/model extraction, smell detection, smell correction, impact analysis, and verification. Among these activities, correction and detection are the most significant ones.

This PhD thesis dissertation only deals with approaches that are directly related to the management of structural issues in object-oriented software. More precisely, this dissertation only addresses problems that can be detected statically in the source code and are related to object-oriented design. A lot of work exists on the identification and correction of different kinds of problems in specific types of software systems, such as databases [BGQR07, JFRS07] and networks [PP07]. These works are outside the scope of this document. The specification and detection of object-oriented design smells is related to the more general field of design pattern specification and detection (*eg.* [GA08]). However, such works are also outside the scope of the study presented in this dissertation.

¹A good reference on software quality factors can be found in the ISO9126 standard [ISO01].

2.1.1 On the notion of the term “*smell*”

First of all, the use of the term “smell” to refer to software design problems must be explained. The interesting connotation of the term “smell” is that it describes a situation where there are hints that suggest there can be a problem. In order to decide whether the problem really exists or not, the situation has to be examined in more detail. The term “smell” can be used to describe a “bad structure” in a program that is evident and obvious, and harmful itself. At the same time, it can be used to describe a design situation that is not so obviously noxious, or a certain software structure that is not disadvantageous by itself, but can indicate the existence of a problem somewhere else in the design. Either way, the term “smell” expresses quite well the shades of gray in the harmfulness that the associated software structure can represent. The term also express well the idea that the design has to be carefully analysed in order to confirm the occurrence of the possible problem. A *Lazy Class* smell, for example, refers to a class that does not hold enough responsibilities. The smell reveals a potential design problem. If the class’ features make sense to be somewhere else, or if the class was added to make room for planned changes that were never made, then the class should be removed. If, on the contrary, the class holds responsibilities that cannot be anywhere else and the benefits of this design are greater than reallocating the class’ features, then the class should be kept as it is.

Since the introduction of the term “code smell” –the original concept related to “smells”– there has been a spread of different terms to refer to a family of very similar concepts. The term “defect” has also been used to refer to software design problems [TSFB99, MGMD08], and the term “flaw” has also been used sometimes [CLMM06, Tri08, Mar01]. In this PhD thesis dissertation, the term “smell” is used, to distinguish it from “defect” or “flaw”, since the latter two terms are associated more frequently with run-time and compile errors. The subsections below are aimed at clarifying the terminology and making a proposal to unify the different terms appearing in the “smell” literature.

2.1.2 Code smells

The term “code smell” was introduced by Kent Beck² to define those structural problems in the source code that can be detected by experienced developers. As written by Kent Beck:

“ A code smell is a hint that something has gone wrong somewhere in your code. ”

The suspect structure may not be causing serious harm (in terms of bugs and failures) at the moment, but it has a negative impact on the overall structure of the system and as a consequence, on its quality factors. Code smells can clutter the design of a system, making it harder to understand and maintain. Moreover, the presence of code smells can warn about wider development problems such as wrong architectural choices or even bad management practices.

The term was presented in [BF99a], where a compilation of “bad smells in code” can be found too. A brief description of the term can also be extracted from the book:

“ ...structures in the code that suggest (or sometimes scream for) the possibility of refactoring. ”

This description states the relationship between code smells and refactorings. Code smells reveal where and how to refactor and, inversely, refactorings become the preferred way to remove code smells.

²See <http://c2.com/cgi/wiki?CodeSmell>

Examples of code smells, to cite just a few, are a field that is set only in certain circumstances –*Temporary Field* [FBB⁺99, page 84]–, a method with a large number of arguments –*Long Parameter List* [FBB⁺99, page 78]– or a subclass that does not use some of the methods and fields inherited from its superclass –*Refused Bequest* [FBB⁺99, page 87]. Another typical example of code smell is the presence of *Duplicated Code* [FBB⁺99, page 76]. This situation appears when a concept –an expression, algorithm, etc.– is used in many places across a software system, because it has not been correctly identified within the design and it has been copied wherever needed. Duplicated code makes it unnecessarily more difficult to maintain the system’s structure because it becomes a source of multiple problems: errors in duplicated code can spread through the system, changes to duplicated code should also be applied to the multiple copies, etc. The code should be refactored in order to unify the duplicated structures into a single definition of the concept they represent. Duplicated code can be the consequence of a design problem, a quick patch to achieve a deadline, or a deep-rooted bad practice of copy-paste reuse.

2.1.3 Design smells

We refer to “design smells” as a concept similar to “code smells” but in more general sense. This term, being proposed here, covers the whole range of problems related to the software’s structure, i.e., the “design” part. Different design smells affect the software at a different granularity level, from methods (eg. *Long Parameter List* [FBB⁺99, page 78]) to the whole system architecture (eg. *Stovepipe System* [BMMIM98, page 159]). This could be one of the reasons why some authors have made further additions to the smell terminology. The design smell concept appears in the literature under a variety of names: design flaws [Tri08, SLT06], disharmonies [LM06], defects [MGMD08], etc. In [Moh08], a distinction is made between smells appearing in Fowler’s refactoring book [FBB⁺99], and in Brown’s antipatterns book [BMMIM98], being identified as code smells and design smells, respectively.

Robert C. Martin refers to “design smells” [Mar03] as higher-level smells that cause the decay of the software system’s structure. He states they can be detected when software starts to exhibit the following problems:

- **Rigidity:** The design is hard to change because every change forces many other changes in other parts of the system.
- **Fragility:** The design is easy to break. Changes cause the system to break in places that have no conceptual relationship with the part that was changed.
- **Immobility:** It is hard to disentangle the system into components that can be reused in other systems.
- **Viscosity:** Doing things right is harder than doing things wrong. It is hard to do the right thing because sometimes it is just easier to do “quick hacks”.
- **Needless Complexity:** The system is over-designed, containing infrastructure that adds no direct benefit.
- **Needless Repetition:** The design contains repeating structures that could be unified under a single abstraction.
- **Opacity:** The system is hard to read and understand and does not express its intent well.

In spite of the different terminology used in the literature, all the problems the authors describe are problems related to a bad design, such as a misidentified or overlooked abstraction, a misused pattern, an under- or over-engineered design, etc. For the sake of simplicity, and in order to unify the existing terminology, we propose to use the term “design smell”. We also propose, for the term, the following definition:

Definition 1. A **design smell** is a problem encountered in the software’s structure (code or design), that does not produce compile-time or run-time errors, but negatively affects software quality factors.

This is the interpretation that will be used in this dissertation, when referring to design smells. The term “smell”, when referring to software’s structure, has widely acquired, and is already being associated with, negative connotations, as in “code smell”. Therefore, we have decided to drop the “bad” adjective from the term “smell” because the negative sense is already implicit in it when referring to “software smells”. The term “design” has been chosen to describe the fact that the referred problems are related to structure, design choices, patterns or formations in the design. We meant “design” much in the sense of software design expressed by Reeves [Ree92]: engineering decisions that take place not only while developing models but while building all the software artefacts, including source code. We refer, therefore, to a variety of structural problems that can spread across different levels of abstraction and refinement. The key difference between design smells and other kinds of software problems lies in the presence or absence of compile-time or run-time errors. That is why the definition explicitly states that errors are not the pernicious effect of design smells, but their negative effect on software quality factors. This negative effect on quality factors can result in a variety of problems such as those cited by Robert Martin. In fact, this negative effect on quality factors could lead to, or boost, the materialization of actual errors in the future.

“Design smell” is a unifying term, covering the whole gamut of problems described in this chapter. Nevertheless, given the different variants of design smell definitions presented in the literature, a brief classification and terminology for them is being proposed.

- **Low-Level Smells:** Smells that can be detected by themselves.
- **High-Level Smells:** Smells which can be composed of other smells. High-level smells can usually be described and detected as a confluence of other smells (low-level ones and maybe high-level ones too).

An example of a low-level smell is *Long Method* [FBB⁺99, page 76], while *Brain Method* can be cited as an example of a high-level smell. According to the definition of *Brain Method* in [LM06, page 92], the *Long Method* smell is one of the indicators used to detect this high-level smell, other indicators being “excessive branching” and “many variables used”. On the contrary, *Long Method* can be detected just by itself, by checking the number of lines of code in a method against a certain threshold.

This dissertation also keeps an orthogonal classification for design smells by catalogue. It happens to be the case that most smells from a specific catalogue fall into one of the previously defined categories, but this will be left open. Most bad smells from [FBB⁺99] are low-level smells, and some disharmonies from [LM06] are low-level smells too. Some disharmonies are high-level smells, while antipatterns belong exclusively to the high-level smells category.

In the following Subsection, a more detailed review of the different design smell catalogues existing in the literature is presented.

2.1.4 Historical background on design smells

Several books relate to design smells. Webster [Web95] wrote the first book on smells in the context of object-oriented programming, including conceptual, political, coding, and quality-assurance pitfalls. Riel [Rie96] defined 61 heuristics characterising good object-oriented programming. These heuristics deal with classes, objects, and relationships, enabling software engineers to assess the quality of their systems manually, and provide a basis for improving design and implementation.

Beck and Fowler compiled 22 code smells that are low-level design problems in the source code of software systems [BF99a]. They present bad smells as hints to look for situations where design problems may exist. These bad code smells suggest where and when the engineers should apply refactorings. Code smells are described in an informal style and associated with a process to locate them through manual inspections of the source code.

Brown *et al.* [BMMIM98] provided another source of smell specifications: the so-called “Antipatterns”. They focused on the whole software development process of object-oriented systems and textually described 41 antipatterns, which are general object-oriented smells. Of these antipatterns, 14 relate to software development (design smells), 13 relate to software architecture, and the remaining 14 to project management. Belonging to the software development category are the well-known antipatterns *Blob* and *Spaghetti Code*. In the present dissertation, we are not interested in architecture and managerial antipatterns since they are quite difficult to detect in source code.

Tiberghien *et al.* [TMMM07] described 47 design smells, most of which are from the other smell catalogues. They use the term “design defects” (*défauts de conception*). The smells listed are all about bad design, except for ‘code level defects’, which address low-level code problems.

These books and catalogues provide in-depth views on heuristics, code smells, and antipatterns aimed at a wide audience for educational purposes. However, manual inspection of the code for searching smells based only on textual descriptions is a time-consuming and error-prone activity. Some works have analysed and compiled design smells with more detailed and structured descriptions to ease their detection and correction.

Kerievsky [Ker04] integrated design patterns with bad smells. He also added some new smells which complement the catalogue from Beck and Fowler [BF99a]. Design patterns are involved with design smells, because some smells can originate from the misuse, or the lack of use, of design patterns. Thus, detection of design smells can be performed through the identification of these situations. Design patterns can also be a tool to correct smells. Kerievsky describes strategies to remove design smells by the introduction of design patterns.

Lanza and Marinescu [LM06] presented a catalogue of design smells called “disharmonies”, along with the definition of *detection strategies* and recommendations for the correction process. This approach introduces metrics-based rules to capture deviations from “good” design principles. The term “disharmony” can be understood more as an attempt to unify the terminology than as a kind of defect on its own. Many disharmonies are similar to bad smells in terms of their abstraction level. Indeed, 7 out of 11 disharmonies are problems already presented in [FBB⁺99]. Three types of disharmonies are described: identity, collaboration and classification disharmonies. The definition of “detection strategies” within the description of disharmonies, is a step towards precise specifications of design smells that can allow automated smell detection. The recommendations for correction, are not so suitable for automation, but if used together with the “detection strategies”, they are sufficiently structured to guide the developer through the correction process.

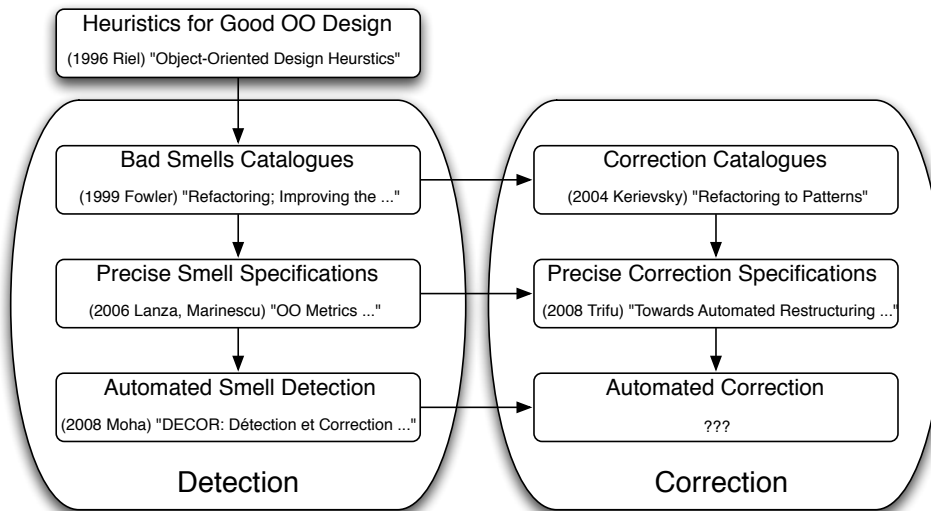


Figure 2.1: *A brief history of design smell management.*

Trifu [Tri08] described a set of 10 design smells, which he coined “design flaws”, and provided “restructuring patterns” to detect and correct them. Restructuring patterns identify not only the smell, but also the intention of the design. This is then used to propose a correction strategy, which is a pseudo-code algorithm based on the application of refactorings and on the introduction of design patterns.

Moha [Moh08] proposed a framework to specify, detect and visualise design smells. She developed a domain-specific language that allows the user to specify smells [MGMD08]. These specifications are transformed into generated Java source code, which can be compiled and run to search for smells. The specification process is manual but the detection and visualisation are fully automated through DECOR, the tool that implements the approach.

From the above, a brief summary of the research history in the field of design smell management is presented in Figure 2.1. Since its early stages, when general guidelines of good object-oriented design practices were provided [Rie96], the field has grown and evolved to propose more automated approaches, not only to smell detection, but to all of the different activities of smell management, such as specification, correction, etc. Regarding correction and detection, the field has evolved through the development of more formal specifications of design smells and correction strategies and, as a consequence, this has also led to an improvement in the automation of both, detection and correction.

Figure 2.1 highlights particular works which we consider to be significant in each improvement step. In the current state of the art, automation for design smell detection has reached a good maturity level, represented by works such as [LM06, Moh08] and their respective prototype tools [iPl, Pti], which demonstrate that there is a chance of seeing industry tools based on them soon. Correction, on the contrary, lacks more successful works on precise and systematic specification and automation. Nevertheless, works such as [Ker04, Tri08] have set the basis for further improving the automation of the smell correction activity. We have observed that the more systematic the approach in specifying design smell is, the more easier to automate the correction of design smells is too.

Two major approaches to design smell management can indeed be distinguished: to prevent smells before they occur or to correct them once they have appeared. These two major approaches are so different in nature that covering both will exceed the limits of this research. Moreover, the former relates to almost any discipline aimed at object-oriented design. The present dissertation will focus on the latter: approaches dealing with design smells that have already appeared in the software artefacts. The proposed approach provides a framework to advance the current state of the art in design smell management, in particular in automated correction supported by the automation of complex refactoring processes.

2.2 Refactoring

Refactorings are structural transformations that can be applied to the source code of a software system to perform design changes without modifying its observable behaviour.

Refactoring is rooted in software restructuring [Arn89, GN93]. Software restructuring is related to any technique and methodology aimed at improving the structure of a software system. More precisely, it is the activity of reorganising the design of a system in order to achieve better modularisation. The changes that the system must accommodate through its unavoidable evolution are the main cause of design degradation. These changes enforce the developer to choose new abstractions and make new design decisions to revert the degradation process, which would result in the restructuring of the system.

Refactoring appeared as the object-oriented version of restructuring. Opdyke's PhD Thesis [Opd92] is considered to be the first work on refactoring. In this work, the technique of restructuring object-oriented programs is given the name of refactoring. The basis of refactoring, which Opdyke introduced, can be summarised as:

- **behaviour preserving invariants:** conditions about the program that are not altered after the application of a refactoring. Behaviour preserving is hard to specify and, thus, to check (specially for non-formal languages), but if it is defined in terms of a finite set of program properties we want to preserve, then behaviour preservation can be checked for them.
- **preconditions:** conditions under which the application of a refactoring can be proved as behaviour preserving. The condition of behaviour preservation can be verified by showing that the refactoring does not affect the behaviour preserving invariants.
- **low-level refactorings:** primitive refactorings aimed at manipulating program entities at the lowest level. They include, for example, creating, removing or renaming different kinds of program entities: classes, methods, attributes, etc.
- **high-level refactorings:** complex refactorings, difficult to put into practice, which can be undertaken by composing the low-level ones. Opdyke also mentions “composite refactorings”, as refactorings also built from low-level ones, but with a degree of difficulty similar to them, in contrast to high-level ones.

The refactoring technique gained popularity since the publication of [FBB⁺99], and its widespread adoption in the Agile Programming Community³. In this book, Fowler *et al.* compile

³A good source of information on agile approaches is at: <http://www.agilealliance.org>

a catalogue of 68 refactorings (and 4 “big refactorings”). Each refactoring definition in the catalogue presents the mechanics of each particular transformation, a short summary of the situation in which the refactoring would be needed, its motivation and a short example. The catalogue of smells included in the book (see Section 2.1.4), helps the developer to decide when and how refactorings should be applied. Using bad smells, they suggest which particular refactorings can be applied for each specific problem, in order to remove or to mitigate the bad smell and to improve the system’s structure. In most cases, different refactorings have to be combined to achieve the desired change.

Applying just a simple refactoring, such as renaming a member attribute, involves not only the local change for that entity, but a number of non-local updates –everywhere the entity is accessed– in order to maintain the references to the renamed member variable and to preserve the program’s behaviour. This repetitive task is tedious for a developer and error-prone, but can be painless and safe if automated.

Widespread usage of refactorings started with the introduction of the first refactoring tool: the SMALLTALK REFACTORING BROWSER, developed by Roberts in his PhD dissertation [Rob99]. This tool assisted the user in the application of refactorings for SMALLTALK programs. Once the refactoring to be applied has been selected by the user, the tool would automatically perform the repetitive and error-prone updates. Refactoring automation is considered the key to a widespread adoption of the technique by software developers [Rob99]. Indeed, this could be said for any technique, but it has been proved to be right with regard to refactorings. An increasing number of tools, from industry and academia, are available to automate the execution of refactoring operations. While refactoring tools were initially stand-alone applications (the SMALLTALK REFACTORING BROWSER⁴, REFACTORIT⁵, BICYCLE REPAIR MAN⁶, etc.), the trend has turned to major IDEs integrating refactoring functionality (INTELLIJ IDEA⁷, ECLIPSE⁸, NETBEANS⁹, etc.). Refactoring support has become a mandatory feature for any software development tool, as it has been shown, for example, in the refactoring survey by Mens and Tourwé [MT04].

Although the automated execution of refactorings is the most mature field related to refactoring support, some works are leading towards other types of improvements. The remainder of this section presents a brief overview of them.

2.2.1 Applying refactorings

As already mentioned, refactoring support is currently present in many refactoring tools. The list of tools that support the automated application of refactorings is quite large. As a small sample, we can refer to the ones previously mentioned: INTELLIJ IDEA, ECLIPSE, NETBEANS, etc. and add more tools to this list, such as: XREFACTORY¹⁰, JFACTORTM¹¹, JREFACTORY¹², JBUILDER[®]¹³, BORLAND®TOGETHER[®]¹⁴, etc.

⁴<http://st-www.cs.uiuc.edu/users/brant/Refactory>

⁵<http://www.aqris.com/display/A/Refactorit>

⁶<http://bicyclerepair.sourceforge.net>

⁷<http://www.jetbrains.com/idea>

⁸<http://www.eclipse.org>

⁹<http://www.netbeans.org>

¹⁰<http://www.xref.sk/xrefactory/main.html>

¹¹<http://old.instantiations.com/jfactor/default.htm>

¹²<http://jrefactory.sourceforge.net>

¹³<http://www.codegear.com/products/jbuilder>

¹⁴<http://www.borland.com/us/products/together/index.html>

Some authors, such as Murphy-Hill *et al.*, are working in this field, researching the improvement of usability in refactoring tools [MHB08, MHB07]. Any approach which improves the refactoring activity, even when it does not offer a fully automated support, is important and relevant to improving the state of the art.

The door is still open for tools integrating more advanced refactoring support. This would enable more complex refactorings, such as the automated introduction of design patterns [GHJV95, Ker04] or the application of “big” [FBB⁺99, chapter 12] or high-level [Opd92, pages 79–148] refactorings.

2.2.2 Developing refactoring tools

Another major field in refactoring support is the development of refactoring tools. Some researchers focus on improving refactoring by studying how to develop better refactoring tools.

Tichelaar *et al.* developed, in [TDDN00, Tic01], a meta-model for object-oriented programs (called FAMIX) that allowed the development of language independent refactoring tools. On top of this meta-model, multiple tools related to refactoring, and more generally to reengineering, have been developed by the MOOSE group¹⁵.

Similarly, the language independent meta-model with support for generics, presented by Crespo in [Cre00], has led to a variety of works on language independent refactoring tools and reusable refactoring components [LMCP06, MLCP07, MC08] within the GIRO group¹⁶.

Works on composition and reuse of refactoring definitions [KK03, KK04] will allow the developer to write and reuse self-defined refactorings and help construct more complex refactorings from simpler ones.

Validation of the refactoring operations implemented by the different development tools is also an important subfield. Automated testing of refactoring executing tools can help to debug and improve them [DDGM07]. Behaviour-preservation of refactoring operations can be checked by executing automatically generated test-suites over the original and the refactored versions of a software system and comparing their results [SGSM10].

2.2.3 Mining refactorings

Refactorings are commonly integrated into development environments and are extensively used. Finding and understanding refactorings is important to document and to understand a system’s evolution. It will be useful to automatically determine when software evolution has been behaviour-preserving: to verify a redesign process, or a handmade refactoring, to find and characterise the stages of a system’s evolution, etc. Finding the refactorings applied to a component will allow these changes to be reproduced in a program that uses it. Therefore, this could help to make the program compatible with the component again after a refactoring process has broken the API of the component.

Some authors have explored the possibility of finding refactorings through metrics analysis [DDN00], or through the computation of lexical distances between elements from different versions of a system [DCMJ06].

We addressed this problem [Pér06, PC07a, PC07b] representing programs with graphs and the refactoring searching problem as a state space search problem. Within this approach, we represented the original and the refactored versions of the system as graphs, and designated

¹⁵<http://moose.unibe.ch>

¹⁶<http://giro.infor.uva.es>

them as the start and goal states of a state space search problem. Refactorings held the role of state changing operations. Within this approach, determining the existence of a refactoring sequence between both versions of the system was tackled as a reachability problem. Finding the refactoring sequence itself was addressed as the problem of finding a path from the start state to the goal state. We applied a graph parsing algorithm in order to perform depth-first searching to find refactoring sequences between two different versions of a software system. This previous work helped us analyse and understand the characteristics of the problem of automating refactoring sequences.

The size of the state space was revealed as the main problem. We concluded that if no conditions are applied to restrict the allowed states or the applicable refactorings, the size of the state space would probably be infinite. If we search for refactorings which can immediately be executed over a system in a specific state, the number is huge, so it is the combinatorial explosion during the searching process. We speculated that if refactoring descriptions were expressed in terms of preconditions, transformations and postconditions, these preconditions and postconditions would guide the search, and the size of the state space could be reduced.

2.2.4 Suggesting refactorings

The last major field in refactoring automation which deserves a small review is suggesting refactorings. Refactorings are transformations aimed at improving the design of a system. A very important area in refactoring automation research is to help the developer, not only in applying refactorings, but also in deciding when and how to refactor. This field is strongly related to the correction of design smells. A developer refactors a system with a purpose in mind. One of these motivations, aimed at the improvement of the system's design, is often to remove, or at least reduce, design smells. One possible way to suggest when and how to refactor in order to improve the system's design, is by showing where and when the design is defective.

Relations between metrics and bad smells have been studied by Mäntylä in [Män03]. In this work, the catalogue of bad smells from [BF99a] is revisited using metrics to establish the inherent relationships between bad smells. The study of dependencies between the different smells allows the author to classify them, proposing a bad smell taxonomy. On the basis of this classification, relations between metrics and bad smells are further analysed in [MCL05, CLMM06]. These works argue that the presence of bad smells may be revealed with metrics, and they explore how bad smells can be paired with the refactorings that may correct them.

Tourwé and Mens presented, in [TM03], an approach to provide an automated support for identifying refactoring opportunities. They used the technique of logic meta programming to detect bad smells and they defined a framework that uses the detected bad smells to propose the refactorings which can improve the system's design by removing them.

Detection of design patterns can also point out when and how to refactor. The work from Kerievsky [Ker04], already mentioned in Section 2.1.4, describes how to use design patterns to guide the refactoring process. The identification of misused design patterns and the detection of structures which could benefit from the introduction of a pattern could help reveal which refactorings have to be applied. The improvement of the system's structure will be performed either by adding a design pattern, consolidating it or removing it.

Melton *et al.* described, in [MT06], a proposal to suggest refactorings that would improve the internal design of a system, making it easier to understand and less error-prone. They detect candidate classes for refactoring by identifying those classes involved in long dependency cycles.

In [MHVG08a], Moha *et al.* use Relational Concept Analysis (RCA) to suggest the best refactorings to improve cohesion and coupling metrics in the process of removing a *Blob* [BMMIM98] from four different open-source software systems.

The work of Trifu [Tri08], already introduced in Section 2.1.4, presents a general framework to develop restructuring patterns. His work focuses on very specific design smells and their possible correction strategies. Although an automated approach to detection is presented, the application of the complex correction strategies is not automated. His restructuring strategies describe very precise algorithms to remove a very specific smell. Nevertheless, the execution of the strategies must be performed by the developer, who may face the problem of unapplicable refactorings, whose preconditions are not met by the system at the very moment of their application. Another work from this author explicitly mentions the inherent difficulty of these complex refactoring processes due to the problem of precondition fulfillment [TR07].

The approach in the present PhD Thesis dissertation presents an automated framework that developers can use to define refactoring specifications and to compile their heuristic knowledge on refactoring mechanics. The developer can add strategies to execute complex refactoring sequences such as introducing design patterns, invoking big refactorings from low-level refactorings, etc. The proposed approach allows the application of these complex refactoring sequences by planning the full sequence ahead and prior to the execution itself.

2.3 Design smell correction with refactorings

The context of this PhD dissertation, which has been overviewed in the present chapter, is summarised in the following paragraphs.

The activity of design smell correction should not change the system's behaviour. When correcting design smells, the objective is not to remove bugs or errors. Hence, concerning the observable behaviour, we aim to leave the system untouched. Once a design smell correction process has been performed, the same outputs should be produced from the same inputs. Thus, the most appropriate technique for software smell correction is a technique that allows the system's structure to be improved without changing its observable behaviour. This technique is, by definition, refactoring [FBB⁺99].

As already mentioned, one of the difficulties in applying refactoring operations is that it is rare for the preconditions of the desired refactorings to be fulfilled by the system's source code in its current state. The failure of some precondition of a particular refactoring, that forces the developer to plan ahead how to solve the problem, is the most frequent scenario. This can be done either by choosing another different refactoring path or by applying other preparatory refactorings to enable the precondition which previously failed. This makes complex refactoring processes even more difficult to undertake.

Providing an approach for planning and generating executable design correction strategies will ease the activity of design smell correction and will improve the state of the art. The correction activity has not been explored in as much detail as the detection one. Most approaches focus on suggesting which are the best redesign changes to perform, and which are the best structures to remedy the smell and reflect the original design intent. The current automation of these correction strategies faces some challenges. The suggestions made for design smell correction by the existing approaches cannot be executed as they are. This is partially due to the heuristic

nature of these correction specifications¹⁷, and partially because of the problems associated with the violation of refactoring preconditions that make these complex refactoring processes hard to overcome.

¹⁷This will be analysed in more detail in Chapter 4.

Chapter 3

A Survey on Software Design Smell Management

This chapter reviews the state of the art in design smell management and presents a survey of design smell management approaches, using feature diagrams as a graphical guidance to illustrate it. This survey, and the taxonomy it defines, can help in various ways. Newcomers in the domain can use it to get acquainted with the important aspects of design smell management, tool builders may use it to compare and improve their tools, and software developers may use it to assess which tool or technique is most appropriate to their needs.

This study has been co-written with Naouel Moha, Tom Mens and Carlos López. A previous version of the survey and the feature-model-based taxonomy presented in this chapter is available as a technical report [PLMM11]. A characterisation of a set of design smell management tools, that follows the taxonomy defined in this previous study, have been performed [MASFPC11]. The results of this characterisation have also been published as a website¹.

3.1 Overview of the survey

This chapter presents the results of a literature study that we have carried out on design smell management techniques. We have also downloaded and evaluated numerous tools that automate some of the design smell management activities. The results of our survey are presented as a concise multi-dimensional classification framework, using feature diagrams as a graphical representation [CE00]. Our framework can be used, among others, to compare the commonalities and variabilities of different approaches and tools based on particular criteria of interest.

Due to the abundance of research literature available on the subject, we have deliberately restricted our study in various ways. First of all, we only consider approaches that are directly related to the management of structural problems that can be detected in software code and design, in particular for object-oriented software. A lot of related work exists on the identification and correction of different kinds of problems in specific types of software-related systems, such as databases [BGQR07, JFRS07], and networks [PP07], but these are outside the scope of this survey. The specification and detection of design smells relate more generally to the field of design pattern specification and detection (*eg.* [GA08]). Such approaches are also outside the scope of this study. Finally, two major approaches to design smell management can be distinguished:

¹<http://www.infor.uva.es/DesignSmells>

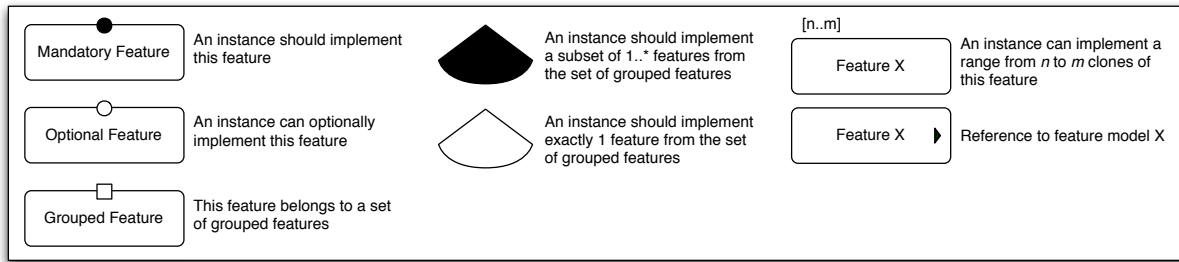


Figure 3.1: Feature diagram notation used throughout this survey.

preventing smells before they occur; and correcting them once they have appeared. These two approaches are so different by nature that a taxonomy covering both of them would not be very meaningful. The survey presented here focuses on the second approach only.

The goal of this study is to present a *survey* of design smell management approaches, providing a framework to compare and analyse the current and future design smell management techniques and tools. This survey can be used for a wide variety of purposes. Among others, it can help software developers choosing a particular approach that is best suited for their needs, it can help tool builders to assess the strengths and weaknesses of their tool compared to other tools, and it can help scientists to identify limitations across tools or technology that need to be overcome by improving the underlying techniques and formalisms.

3.1.1 Feature modelling notation

As a visual aid to guide our design smell management survey, we rely on a visual notation called feature diagrams, that is inspired by the one used by Czarnecki and Helsen to present their survey on model transformation [CH06]. *Feature diagrams* are a visual representation of feature models. Different symbols and notations for feature diagrams can be found in the literature. The notation used in this study is shown in Figure 3.1.

Feature modelling [CE00] is the activity of modelling the common and variable properties of concepts and their interdependencies by organising them into a coherent model referred to as a *feature model*. This model is used to represent a hierarchy of features, representing the common and variable properties of concept instances and the dependencies between the variable features.

Feature models are a nice and intuitive way to represent a family of systems, or a concept such as design smell management, through the analysis of the commonalities and differences between the wide variety of approaches supporting it. Features are important properties of a concept, and the aim of feature modelling is to describe a family or a concept, within a given domain, through the analysis and specification of its particular occurrences. Features also serve to capture and model the knowledge and terminology of that domain. The basis for feature modelling can be found in Czarnecki and Eisenecker's book [CE00].

During the analysis of the surveyed design smell management approaches and tools, we have detected many commonalities shared between all of them, or between some subsets. For example, all approaches address a certain type of target artefact. In the same way we have identified relevant differences that can be used to characterise each approach. As an example

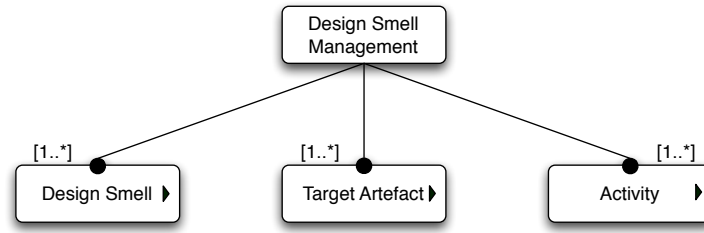


Figure 3.2: Top-level variation points for comparing design smell management approaches.

(see Section 3.3), an approach can search for design smells in source code, or in executable code, models, etc. Consequently, for this survey, we have found feature diagrams to be an appropriate and useful notation. Moreover, since all design smell management tools belong to a “family” of approaches, they can be naturally described with feature diagrams.

The reader should note that this survey, and the feature diagrams that illustrate it, reflect the current state of the art. Therefore, the features identified in some categories, such as “Smell Property” in Section 3.2 or “Type of Representation” in Section 3.3, may be extended by future approaches. For this reason, we have sometimes added to the feature diagrams elements that do not match current approaches, but rather illustrate what we consider to be feasible or desirable in future approaches. Such is the case of the “Target Artefact” multiplicity in Figure 3.2.

3.1.2 Top level features of the design smell management survey

The proposed feature diagram notation allows us to group design smell management activities, tools, techniques or formalisms based on their commonalities. We adopt a multi-dimensional classification, allowing us to describe and compare different approaches, based on the criteria of interest. Within our classification, we do not consider general properties such as interoperability, usability, or extensibility because these are tool-specific properties that can apply to any kind of tools regardless of the domain of interest. As such, they are not specific or intrinsic to design smell management approaches *per se*.

To present our survey in a structured way, each of the following sections discusses the main features with respect to design smell management that can be used to group together approaches sharing these features. We start by describing the top level features, constituting the main common properties of our field of study. We then descend down the model to describe each of the subfeatures. The root feature of our model is *Design Smell Management*. It represents any approach dealing with design smell management. An instance of the model will represent an incarnation of an existing approach, or even a non-existing one that would be feasible and interesting to develop. The three top-level features of the *Design Smell Management* root feature are shown in Figure 3.2 and explained below. These features describe properties common and mandatory to every design smell management approach.

Design Smell. A wide range of design smells can be managed by different approaches. The nature of the smells each approach addresses is a major top-level variation point.

Target Artefact. Any design smell management tool requires at least one software artefact on which the smell can be observed. We will refer to this (set of) software artefact(s) as the *Target Artefact* of the approach.

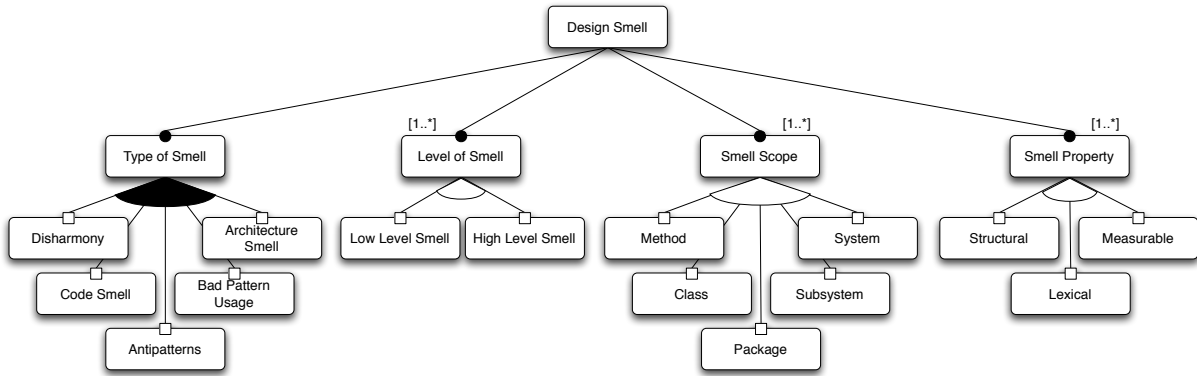


Figure 3.3: Top-level Design Smell feature, and its subfeatures.

Activity. The third top-level variation point for design smell management approaches is the set of activities explicitly supported by each approach. An example activity is smell specification. Every approach requires a definition of design smells in order to be able to detect them, but only some approaches present explicit support for the smell specification activity.

The next three sections of this chapter will discuss each of these three main features in detail, by further decomposing them into subfeatures and using them to survey the state of the art in design smell management.

3.2 Design smell

As introduced by the root feature diagram in Figure 3.2, a design smell management approach can cover several smells. The *Design Smell* feature, depicted in Figure 3.3, allows to describe in more detail the nature of those smells. This feature is split into five branches of mandatory features. Most approaches are specialised in: (1) a certain type of smells, (2) smells at different levels of abstraction, (3) smells affecting several program entities at a particular level of granularity, and (4) smells that have internal properties of different natures. This classification of design smells extends those proposed in [MLC06, MGDLM10]. Although we use it for the purpose of establishing a classification framework for design smell management approaches, it is also useful on its own, for classifying design smells themselves.

Type of Smell. This feature describes the type of design smells addressed by an approach. By “type of smell” we refer to the catalogue in which the smell is defined. Some design smell management approaches define their own set of smells [LM06], but the vast majority of them are focused in those smells described in a small number of catalogues. Those represented in the feature diagram of Figure 3.3 are the most widely referenced catalogues. To characterise a design smell management approach, we should describe which catalogue(s) of smells it addresses.

Marinescu *et al.* [LM06] define *disharmonies* as design smells that affect single entities such as classes and methods. The particularity of these disharmonies is that their negative effect on the quality of design elements can be noticed by considering these design elements in isolation. They identify three aspects that contribute to identify disharmony of a single entity: its size, its

interface and its implementation. IPLASMA [iPl], INCODE and INFUSION [Int08] are three tools that detect these disharmonies, using object-oriented metrics with customized filters.

Most of the studied approaches focus on those code smells specified in the catalogue of Fowler *et al.* [FBB⁺99], such as [AS09, CLMM06, TCC08, Mun05, Sem07]. Munro [Mun05] proposed metric-based heuristics with thresholds to detect code smells, which are similar to Marinescu’s detection strategies [Mar02]. Alikacem and Sahraoui [AS09] proposed a language to detect smells and violations of quality principles in object-oriented systems. This language allows specifying rules using metrics, inheritance, or association relationships among classes, according to the user’s expectations. It also uses fuzzy logic to express the thresholds of rule conditions.

The DECOR method developed by Moha *et al.* [MGDLM10] focuses mainly on Brown’s *antipatterns* [BMMIM98]. As another example, ANALYST4J [Ana08] allows the identification of antipatterns and code smells in JAVA systems using metrics.

Some approaches use *design patterns* to search for design smells [GHJV95], either to look for opportunities to apply a design pattern or to detect misapplication of patterns [GAA01, Ker04, TM03]. Guéhéneuc *et al.* [GAA01] use design patterns as reference structures, and detect design smells as failed intents of applying design patterns. For this, they search for structures that resemble design patterns but slightly deviate from them. To correct these smells, they suggest to transform these structures so they match the intended design pattern properly. In [Ker04], Kerievsky proposes a more manual approach for detection and correction of design-pattern-related smells. He instructs the developer to search for structures that reveal that a design pattern has been misapplied, but he also describes situations where a design pattern is absent and can be introduced, and situations where a design pattern is cluttering the design and therefore, should be completely removed.

Roock and Lippert [SR06] present a catalogue of ‘architecture smells’ related to the organization of subsystems. Among these smells we can find *dependency-related problems* such as cyclic dependencies. Such smells can be detected by tools like [Con99, STA, Tes08].

Level of Smell. Another way to classify design smells is through the distinction between low-level and high-level smells. *Low-level smells* focus on a single very specific problem. Code smells [FBB⁺99] such as are “long methods”, “data classes”, and “large classes” are examples of low-level smells that refer to very specific situations observed in the code. *High-level smells* are design smells that are composed of other smells, such as antipatterns. They focus on a variety of similar (but different) problems. An example of a high-level smell is the Blob antipattern [BMMIM98], also known as God Class. It reveals a procedural design (and thinking) implemented with an object-oriented programming language. It manifests itself through a large controller class that plays a God-like role in the program by monopolizing the computation, and which is surrounded by a number of smaller data classes providing many attributes but few or no methods. This high-level smell is composed of other low-level smells such as the code smells “data class” and “large class”.

Smell Scope. This feature is used to describe the extent or scope of the different types of entities involved in the supported smells. For most object-oriented languages, the different scopes would be system, subsystem, package, class, method and statement. High-level smells usually represent design problems with a wide scope, affecting several and/or large entities. Low-level smells, on the contrary, have an effect over a well-defined and limited scope, *i.e.* within a single and small entity. As illustrated by the multiplicity associated to this feature, a design smell can extend over several entities belonging to different levels.

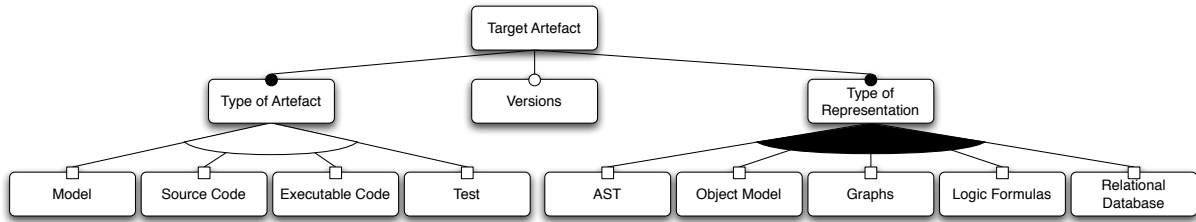


Figure 3.4: Top level Target Artefact feature, and its subfeatures.

Smell Properties. The nature of a code smell can be summarised and represented by the smell indicators or *properties* used in its specification. These properties can be decomposed into: *structural* descriptions of “smelly” structures in the design of code; *measurable* specifications based on metrics and measurable properties of the system; and *lexical* definitions based on the names of the software entities. Design smell management approaches benefit from those properties in order to tackle a particular activity. For example, in the Blob description, the large class corresponds to a measurable property that can be easily computed by counting the number of methods and attributes, whereas the data class is a structural property consisting of identifying accessor methods. A lexical property in the Blob corresponds to the use of procedural names (such as Main, Make, Create, Exec) used in the classes affected by such smell. iPlasma [iPl, LM06] uses the measurable properties of a Blob – referred to as God Class – to detect it, while DECOR [MGDLM10, Tea09] additionally uses the lexical property to perform the detection.

3.3 Target artefact

The *Target Artefact* feature is a major variation point to distinguish design smell management approaches. This feature refers to the software artefacts on which the smells can be observed. It is shown in Figure 3.4, and its subfeatures are presented below.

3.3.1 Type of artefact

Any approach addressing the management of design smells, should focus on, at least, one type of software artefact. The types of artefacts supported by an approach, and the way they are represented internally, are tightly coupled to which smells can be managed and how.

A particular tool or technique often targets only a single type of artefact. In fact, we could not find a tool that supports many types of artefacts. Nevertheless, it is feasible and desirable to build a tool that uses different types of artefacts as complementary sources of information, thereby improving its results. The manual detection process described by Travassos *et al.* [TSFB99] illustrates the feasibility of this. To identify code smells, the developer is instructed to examine different types of models, such as requirements descriptions, use cases, class diagrams, class descriptions, state diagrams, and interaction diagrams. This is shown in Figure 3.4 as a decomposition of feature *Type of Artefact* into a set of OR grouped subfeatures. The most common types of artefacts are source code and bytecode or executable code. In addition, several approaches support detection of design smells by analysing software models. A tendency of modelling tools

is, for example, to provide model warnings, so-called “critics”, to the user. This is done by ARGOUML [Arg] and TOGETHER [Bor].

The vast majority of the available tools are aimed at managing source code smells. For example, CHECKSTYLE [Che04] and HAMMURAPI [Ham07] load JAVA code and search for violations of coding standards. The concept of design smell is language-agnostic, so many other programming languages may be supported as well. For example, REEK [Rut] and ROODI [Roo] search for design problems in RUBY source code. Some authors, such as Ciupke [Ciu99] and Sahraoui *et al.* [SGM00] analyse smells in C++. FXCOP [FXC06] and STYLECOP [Sty] find deviations from code conventions in C# code. iPLASMA [iPl, LM06], DECOR [MGDLM10] and INCODE-INFUSION [Int08] provide multi-language support for the analysis of C++, JAVA and C# programs.

Another widely supported type of artefact is executable code –*eg.* binary code or bytecode. Tools such as REVJAVA [ReJ] and STAN4J [STA] analyse JAVA bytecode. The advantage of addressing executable code is that these tools can be used even when the source code is not present.

An emerging trend in many software fields, and especially in *agile software development*, is to treat *tests*, and more precisely scripted tests [Mes07], as first-class citizens. An increasing number of authors have addressed the problem of smells in scripted tests [Mes07, vDMvdBK01] and propose approaches that provide support for managing design smells in scripted tests [NB07, VRDBDR07].

3.3.2 Versions

A design smell management approach can benefit from the additional information that can be extracted from a version repository that stores multiple versions of the target artefact(s) under study. Some code smells from [FBB⁺99], such as *Shotgun Surgery*² are more easily identified by analysing the change history of the system. Support for multiple versions of an artefact is an optional subfeature that it is applicable to any type of artefact.

Several approaches [GDMR04, RSG08, XS04] propose to use different versions of the target artefact as input. Their supporting tools often include support to access version repositories (*eg.* CVS, SVN, GIT, etc.), and to extract and analyse the software artefacts and their metadata from these repositories. Some of these approaches [GFGP06, LWN07a] even claim that this is the only way to detect some particular smells or to obtain a wider picture of a certain problem.

3.3.3 Type of representation

Every design smell management approach is based on an internal representation of the software artefact the approach is dealing with. In order to analyse and process the targeted artefact, its internal representation will be used. This feature is relevant because there is a strong dependency between the internal representation and other aspects such as the technique, the expected results or the automation support.

A common way of representing a software artefact is by means of an *Abstract Syntax Tree* (AST). This type of representation is especially frequent in approaches that target source code or executable code. A typical AST representation will keep the complete information available

²Whenever a change has to be made to a part of the system, many more little changes to other parts of the system are needed too.

in the examined artefact. Different approaches [TCC08, Sli05, TK04, TCC08] can simplify the AST to keep just the relevant information needed for the task at hand, or even augment the AST with additional details in order to ease the design smell management activities.

Other tools rely on a representation based on some *Object Model*. They use a specific meta-model of the targeted artefact, such as MOON [Cre00, CLMM06], FAMIX [Tic01, LM06] or MEMORIA [Rať04, iPl] This type of representation is mostly used to simplify the target artefact. Just the information needed by the approach is extracted from the targeted artefact. Analysis and manipulation of the targeted artefact is performed by programmed procedures using a wide variety of programming languages, even custom-built Domain-Specific Languages (DSL) [Gué03, MGLM⁺09].

Graph-based approaches can also be employed to analyse the target artefact in search of design smells [EM02]. Graph theory can help analyse artefacts in search of defects, and graph transformation techniques to apply corrections [MVDJ05]. De Lucia *et al.* [DLOV08] represent classes as graphs and use measurable properties of these graphs to find refactoring opportunities that will improve the cohesion of that classes.

Logic Formulas offer a similar formal support to represent software. This type of representation enables the use of logic-based techniques to manage design smells [Ciu99, TM03].

Some tools store the information extracted from the targeted artefact in a *Relational Database*. This usually speeds up the task of querying the software model by taking advantage of a dedicated and specialised query engine. This type of representation can be used just internally by a tool [GFGP06, SSL01, TSG04], or even be offered to its final users, enabling them to analyse the artefact by means of structured and composable queries. For example, with SEMMLECODE [Sem07], engineers can execute queries against source code, using a declarative query language called .QL, to detect code smells.

These different types of representation can be combined. An artefact represented as an AST can be analysed with graph algorithms, if it is formalised with graphs, or logic formulas. A model can be stored in a relational database in order to provide an easy and efficient way to access it by querying the database.

3.4 Activity

The *Activity* feature, depicted in Figure 3.5, represents a major variation point among design smell management approaches. The design smell management process can be decomposed into different types of activities that can be supported by a particular approach. We have decomposed this feature into 5 types of activities that we have found in all approaches we have surveyed. For each type of activity we describe the *techniques* used to support it, the *automation support* achieved and the type of *results* that the different approaches produce.

The *automation support* of an activity reflects the maturity of the studied approach in supporting this activity. We distinguish between the following levels of automation, depicted in Figure 3.6, which simplify the ones defined by Sheridan [She00]:

Manual: The activity is carried out in a manual way.

Suggest Alternatives: The tool can execute the activity automatically and suggest options or alternatives to the user. The user still needs to select and apply the suggestion manually.

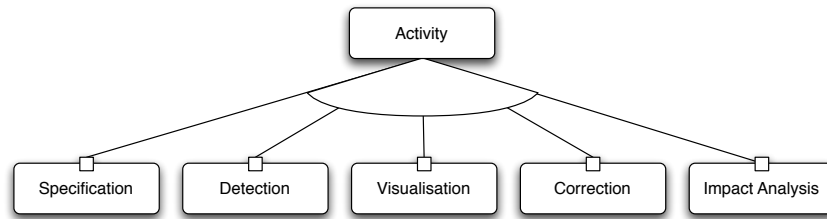


Figure 3.5: *Top level Activity feature, and its subfeatures.*

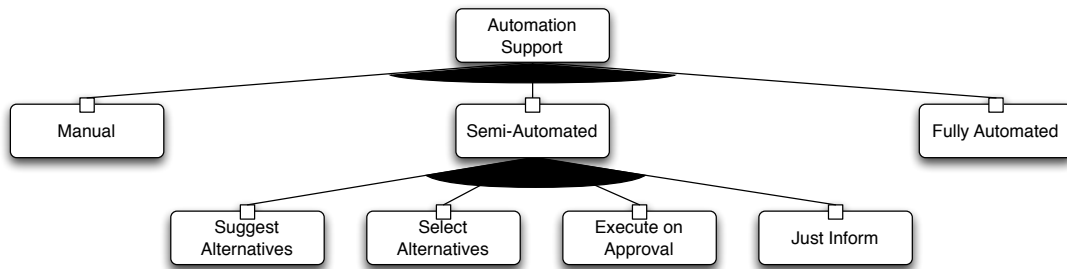


Figure 3.6: *Degree of automation support of a design smell management activity.*

Select Alternatives: The tool suggests and selects the alternative tasks to be performed. The user needs to confirm this selection.

Execute on Approval: The tool presents the user the activity that is going to be executed, but requests permission. The user can only choose to apply the activity as a whole, or to cancel it.

Just Inform: The tool decides and executes the activity without asking the user, but informs the user about the process.

Fully Automated: The tool performs the activity in a fully automatic way, without informing the user of what is happening.

Any design smell management approach must produce some results when applied to a software artefact. To compare approaches, we found it necessary to describe the type of *result* that is being, or can be, obtained from each activity. Analysing an approach or a tool according to this dimension is important in order to determine whether it is useful to solve a particular problem, or whether it can be combined with another approach or tool.

In each of the following subsections, we survey the current state-of-the-art of supporting a particular design smell management activity, in the level of detail that we have explained above.

3.4.1 Specification

The *specification* activity is implemented by those approaches that provide the developers the necessary support to extend or adapt it to their particular needs, by specifying new design

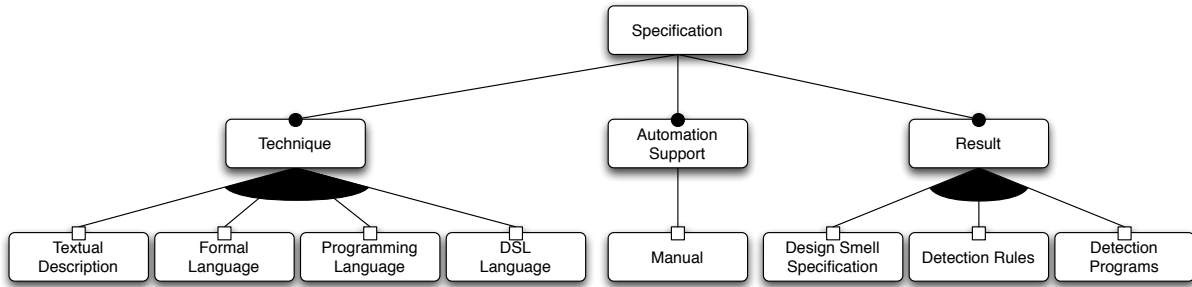


Figure 3.7: Summary of the current situation for the specification activity.

smells or modifying existing smells. Figure 3.7 summarises the techniques, automation support and results for the specification activity.

Technique. Most of the surveyed approaches that support the activity of specification include a description, at least textual, of the design smell for detecting or correcting it. Typically, the technique used by tools to specify design smells is through the use of (possibly customisable) rules expressed in some *formal language* (eg. OCL, SQL, XPATH, logic formulas), *programming language* (e.g. JAVA), or *domain-specific language* ([MGLM⁺09]). The design smell specifications in [FBB⁺99], include a description of general guidelines to correct them. Wake [Wak03] specifies these correction guidelines in the form of recipes. The antipatterns book [BMMIM98] provides a specification in terms of counter-examples.

Automation. Several approaches or tools allow writing user-defined detection or correction rules such as PMD [PMD02], ECLIPSE’s METRICS plugin [Ecl], REFACTORIT [Aqr02], iPLASMA [iPl, LM06], DECOR [MGDLM10] and CODERAIDER [Dur07]. However, this activity is intrinsically manual. The ability to define or tune the smell detection rules is common to most approaches that deal with smells based on metric warnings. SEMMLECODE [Sem07] provides a quite versatile interface to specify design smells, through its built-in query language that can be used to write complex queries to detect smells.

Result. The *specification* activity produces either a purely descriptive and non automatable *design smell specification*, or some kind of *design smell detection rules* that can be automated or even consists of an executable *detection program*.

3.4.2 Detection

The vast majority of existing design smell management approaches focus on the activity of *design smell detection*. Figure 3.8 summarises the techniques, automation support and results for the detection activity.

Technique. One of the approaches to detect design smells is through *manual code inspection* [TSFB99]. Most detection approaches, though, are based on the use of *metrics*. The vast majority of metric-based approaches rely on *structural metrics* [CK91], such as [BEG⁺06, LM06, SLT06] but some recent approaches are taking into account *semantic metrics* as well [DLOV08, ED00]. Structural metrics correspond to metrics derived from syntactic aspects of object-oriented code,

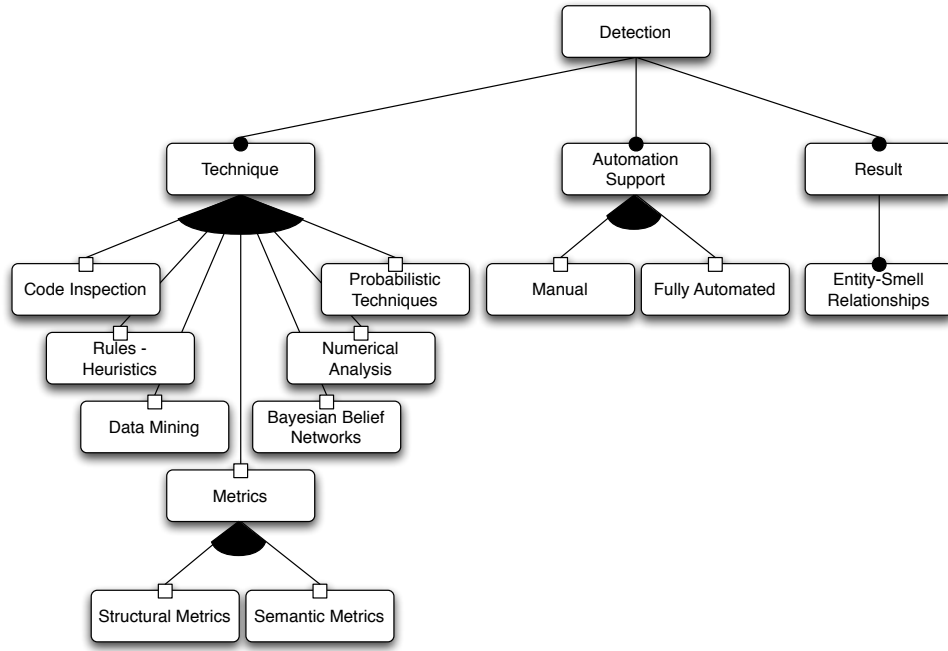


Figure 3.8: Summary of the current situation for the detection activity.

such as the analysis of relationships among the methods and attributes of a class. The metrics defined by Chidamber and Kemerer [CK91] such as Depth of Inheritance Tree (DIT), Lack of Cohesion of Methods (LCOM), and Coupling Between Objects (CBO) are typical examples of structural metrics. Semantic metrics are based on the analysis of the semantic information embedded in the code, such as comments and identifiers [ED00]. Knowledge-based, program understanding, and natural language processing techniques are used to compute such metrics. For example, the semantic LORM (Logical Relatedness of Methods) metric [ED00] measures the cohesion of a class, and more precisely the conceptual relatedness of the methods of the class, as determined by the understanding of the class methods represented by a semantic network of conceptual graphs. Several approaches use *rules or heuristic* knowledge to detect design smells [Ciu99, KRW07, LM06, MGDLM10]. Some approaches resort to more advanced techniques coming from the field of artificial intelligence, such as the use of *data mining* techniques [XS04]; from the probabilistic field, such as *Bayesian belief networks* [KVGS09]; or from *numerical analysis*, such as B-Splines [OKAG10]. However, these techniques may be insufficient to detect some code smells such as Shotgun Surgery [FBB⁺99] and Divergent Change [FBB⁺99], where the design change propagation probabilities between artefacts have to be considered when an artefact changes. Rao *et al.* [RR08] proposed a quantitative method for detecting these code smells using a *design change propagation probability* matrix.

Automation. Some of the proposed approaches are theoretical, aiming to get a scientific understanding of the intrinsic difficulties involved in detecting design smells. Other approaches are fully manual, such as the use of manual code inspection techniques to find design smells [TSFB99]. Most of the surveyed approaches, however, provide explicit tool support, as is the case for [Chi02, iPl, MLC05, MGDLM10, Sli05, Tri08, TCC08, WP05].

Result. The detection activity produces, in all the approaches we surveyed (for example in INCODE-INFUSION [Int08]), lists of *entity-smell relationships* for each detected smell. Those results are presented in a variety of textual and graphical ways.

3.4.3 Visualisation

The *visualisation* activity produces some kind of graphical representation of the target artefact, allowing quick and easy identification of some of its properties. Many approaches address the visualisation activity in the context of design smell management, mostly to present detected smells in a graphical way. A visualisation tool can also provide other kinds of information. It can help the developer to decide which are the best modifications in order to remove a given smell. It can be used for evolution or for explaining the causes and impacts of smells. Visually summarising the properties and characteristics of the system and its parts can ease the realisation of any other activity. Figure 3.9 summarises the techniques, automation support and results for the visualisation activity.

Technique. The type of visualisation technique used mainly depends on the type of information that needs to be visualised. If this information is essentially a spreadsheet table, one can visualise it as pie *charts*, bar charts, line charts and the like. If the information is essentially a *graph*, one needs graph-based visualisation techniques and more or less sophisticated graph layout algorithms. This is the case, for example if one needs to represent (part of the) software structure, such as the dependency relationships [Con99, Tes08]. Some approaches [LM06] define new visualisation techniques, such as the *Overview Pyramid*. It is a metrics-based means to both describe and characterise the structure of an object-oriented system by quantifying its complexity, coupling and use of inheritance.

Automation. The activity of visualisation typically requires human intervention, either during the construction of the visual representation (e.g. by selecting the areas of interest, or choosing the most appropriate visual representation or layout algorithm) or during its use. If the visualisation is static, it can only be viewed [Con99, Tes08]. If it is dynamic, there is typically a direct and explicit link back to the target artefact under study (eg, INCODE-INFUSION [Int08]). This facilitates and speeds up detection and correction of design smells.

Result. Many of the surveyed approaches support some kind of visualisation by producing different types of diagrams to offer different types of general artefact visualisation in order to assist the comprehension of the target artefact. For example, in DECOR, systems are represented as class diagrams and classes infected by smells are highlighted in red. Other kinds of visualisation aids in design smell management include *entity-property visualisation* and *design smell visualisation* [EM02] such as Overview Pyramid and Polymetric views [LM06] or dependency graphs [Con99].

3.4.4 Correction

Approaches supporting the *correction* of design smells, provide a way to (suggest how to) modify the target artefact, in order to remove smells and improve its design. During our survey, we found considerably less approaches supporting a (partially) automated correction of design smells, so this activity is clearly less mature than the smell detection or visualisation activity. Figure 3.10 summarises the techniques, automation support and results for the correction activity.

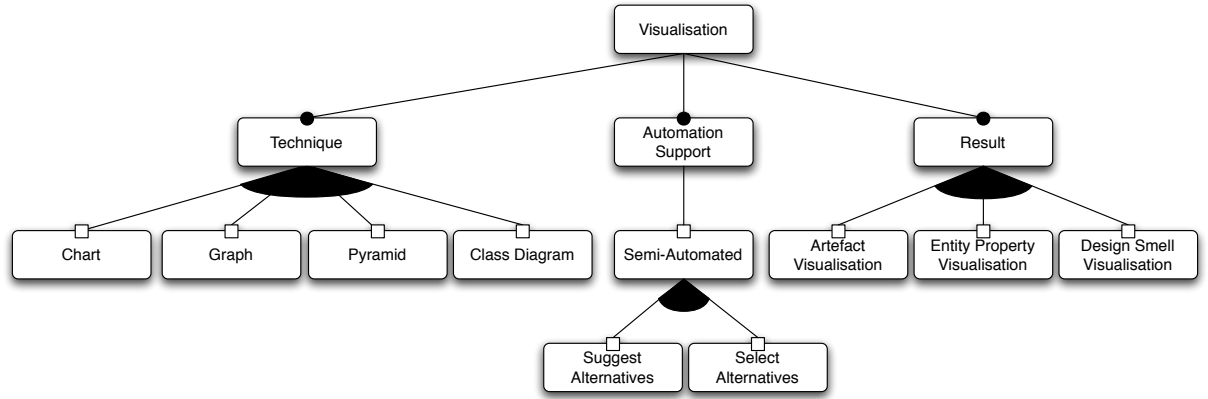


Figure 3.9: Summary of the current situation for the visualisation activity.

Technique. During our survey we have encountered mainly four different techniques for correcting design smells. The first technique is to apply *rules* or *refactoring strategies* to design smells that have been previously detected [MGDLM10, TM03, TSG04]. These approaches have the advantage to provide a comprehensive process both for the detection and correction of smells. The second category suggests corrections (called *refactoring opportunities*) by relying only on metric values or the presence of certain patterns, without explicitly identifying design smells [BCT07, DLOV08, GAA01, SGM00, SSL01, SS07a, TK04]. The third technique relies on ideas coming from the *machine learning* field, where *genetic algorithms* and other types of automated learning are exploited [BCT07, BAMN06]. Formal Concept Analysis (FCA) [GW99] is a fourth technique that has been intensively investigated for restructuring class hierarchies [ADN05, SLMM99, ST00] and classes affected by smells [MHVG08b].

Automation. Various tools support smell correction in a semi-automated way [BEG⁺06, TSG04, TCC08], although all of them need some additional interaction by the user.

Result. The correction of design smells can produce different types of results depending mostly on the degree of automation provided by each particular tool. Some tools can provide just *correction suggestions*, while others can produce some kind of *correction plans*, or specifications of the transformation sequence needed to improve the target artefact’s design. For example, Trifu *et al.* [TSG04] proposed correction strategies mapping design smells to possible solutions. However, a solution is only an example of how the program *should have been implemented* to avoid a smell rather than a list of steps that a software engineer could follow to correct the smell.

Some tools can apply the computed changes to their internal artefact representation, therefore, producing a *transformed artefact model*. The fully-automated tools can operate straight over the target artefact, generating a *transformed artefact*.

3.4.5 Impact analysis

The activity of *impact analysis* refers to the ability of an approach to compute the change impact of a design smell [EM02, VRDBDR07] or the actions performed to remove it [FTC07, LWN07b, TSG04]. Figure 3.11 summarises the techniques, automation support and results for the impact

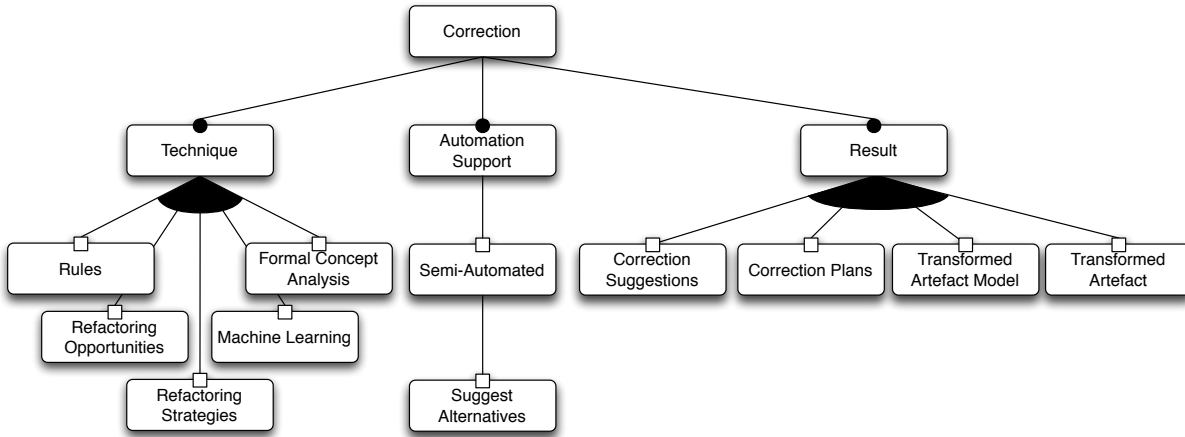


Figure 3.10: Summary of the current situation for the correction activity.

analysis activity.

Technique. Approaches based on quality models offer this feature [BAMN06, LM06, RSS⁺04, SGM00, TK04]. In [TK04], Tahvildari *et al.* present an approach based on soft-goal models. With soft-goal models they define the effects of design smells over metrics and system quality factors in a way that this information can be used automatically by a detection or a correction tool. Using soft-goal models the impact of design smells can be automatically computed to assist the detection and correction activities. Marinescu introduces in [Mar02] how quality models can be used to estimate the impact of a design smell.

Deligiannis *et al.* [DSA⁺04] presented a controlled experiment on the impact of design smells, in which they studied 20 subjects to evaluate the impact of God Classes on the maintainability and understandability of object-oriented systems. The results of their study show that the Blob antipattern affects the evolution of design structures and the subjects' use of inheritance. Other similar approaches based on controlled experiments studied the impact of design smells on software quality factors such as comprehensibility [DDV⁺06] and maintenance [OCBZ09]. Some approaches used statistical models to investigate the relationship of design smells with class error probability [LS07] or with change-proneness [KPG09].

Recent works have studied the impact of design smells on software evolution by analyzing several versions of software systems [OCBZ09, VKMG09]. These approaches identify mainly evolution patterns of smells, which are then used to explain the impact of smells on the rest of the system. For example, Olbrich *et al.* [OCBZ09] analysed the historical data over several years of development of two large scale open source systems. They concluded that God Classes and Shotgun Surgery have a higher change frequency than other classes; and thus, may need more maintenance than non-infected classes.

Automation. Most of approaches evaluate the impact of design smells manually by conducting controlled experiments with subjects [DSRS03, DSA⁺04, DDV⁺06]. Some approaches integrate this activity in a fully automated way [TK04] to assist the detection and correction process.

Result. During our survey, we found that the most common results of impact analysis are the

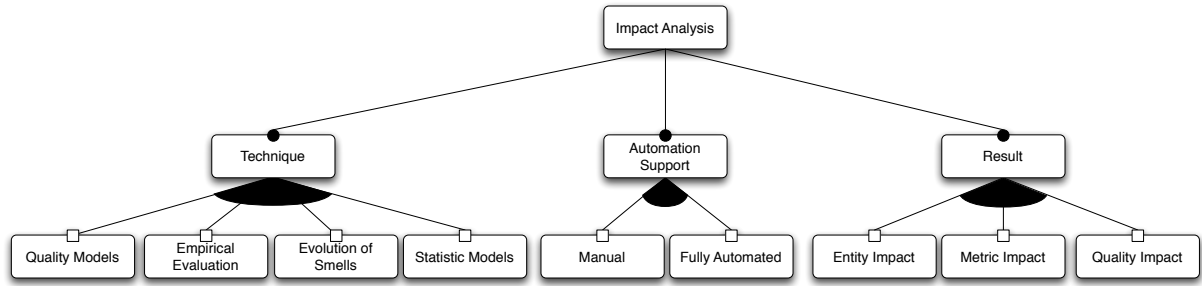


Figure 3.11: Summary of the current situation for the impact analysis activity.

list of entities affected by a smell (*Entity Impact*) and the effect that a smell, or its removal, has over a metric value (*metric impact*) [DSRS03] or over a quality factor (*quality impact*) [DSA⁺04, DDV⁺06].

3.5 Conclusions of the survey

During our survey we have observed that not many of the studied approaches allow to reason about design smells at the level of design models (as opposed to source code and executable code). With the ever increasing importance of model-driven software engineering, it is imperative to have better future support for design smell management at the modeling level.

Related to the above challenge is the fact that almost none of the surveyed approaches is able to deal with design smells that involve different types of artefacts (e.g. a smell that involves both code, models and tests). Support for such types of smells will continue to gain importance, in the presence of multi-view and multi-language development environments.

Only a small fraction of the surveyed approaches took into account the version history. Such a rich data source is able to provide much more relevant information about why a particular design smell occurs, and how it may be corrected. As such, this can give rise to better and more reliable design smell detection and correction tools.

Most of the approaches we surveyed that support the activity of design smell correction were research prototypes. The next generation of commercial design smell management tools should therefore strive to integrate and automate correction techniques, rather than only supporting the detection activity.

Another challenge is to come up with better and more *language-agnostic* design smell management approaches. Most of the surveyed approaches focus on a specific programming language. A few of the tools, though, are applicable on more than one language.

As a final challenge, while the main purpose of current-day design smell management approaches is to improve the *software product quality* (by detecting and correcting “smelly” parts of the software), they could also be used to improve the *software process quality*. In many situations, the cause of a design smell may be a suboptimal software process. (For example, if the software process does not discourage copy-paste reuse, the software is likely to suffer from code duplication and high coupling between modules. Similarly, if the process does not encourage modular design, the software is likely to suffer from cyclic dependencies.) Hence, the smell cor-

rection activity should not only suggest to correct the detected problem itself, but also its cause, by providing concrete suggestions on how to improve the software process to avoid introducing design smells before they occur.

Chapter 4

Refactoring Strategies

This chapter reviews the current situation on design smell correction and performs a domain analysis on the subject. It presents models that describe how this problem is supported and how this support could be improved by means of the computation of complex refactoring sequences. Refactoring strategies are thus proposed and defined as a way to tackle the problem of planning complex refactoring sequences ahead. Finally, the requirements to provide support for refactoring strategies and the characterisation of this problem are presented.

4.1 Analysis of design smell correction specifications

In this Section we perform a domain analysis on the current state of design smell correction and we present a model for it as a result. This model has been written after analysing how different authors have tackled this problem. Authors have addressed the design smell correction activity by compiling empirical knowledge and expertise into rules. Most of these rules have been written in natural language and in an informal way, so they should be used by an experienced developer. In order to analyse how correction procedures are, and should be, supported by a tool, this Section presents a review of the different correction specifications that can be found in the literature. It would be possible to transfer these rules to an automatic tool with different degrees of success, depending on their level of detail.

4.1.1 Current design smell correction specifications

The sources we have analysed are the antipatterns book from Brown *et al.* [BMMIM98], the bad smell catalogue in the book from Fowler *et al.* [BF99a], the refactoring examples in the book from Wake [Wak03], the pattern-related refactorings in the book from Kerievsky [Ker04], the disharmonies catalogue in the book from Lanza and Marinescu [LM06], the reengineering patterns catalogue in the book from Demeyer *et al.* [DDN08] and the correction strategies in the PhD Thesis dissertation of Trifu [Tri08]. To illustrate the result of this domain analysis, we have selected the same design smell – *Large Class*¹ – so it is easier to compare the different “styles” and levels of detail among the different authors. The correction specifications from these authors

¹Depending on the author, we can find this smell under different names and with slight variations in its definition: *The Blob*, *God Class* or *Schizophrenic Class*.

“ As with most of the AntiPatterns in this Section, the solution involves a form of refactoring. The key is to move behavior away from the Blob. It may be appropriate to reallocate behavior to some of the encapsulated data objects in a way that makes these objects more capable and the Blob less complex. The method for refactoring responsibilities is described as follows:

1. Identify or categorize related attributes and operations according to contracts. These contracts should be cohesive in that they all directly relate to a common focus, behavior, or function within the overall system. For example, a library system architecture diagram is represented with a potential Blob class called *LIBRARY*. In the example shown in Figure 5.3, the *LIBRARY* class encapsulates the sum total of all the system’s functionality. Therefore, the first step is to identify cohesive sets of operations and attributes that represent contracts. In this case, we could gather operations related to catalog management, like *Sort_Catalog* and *Search_Catalog*, as shown in Figure 5.4. We could also identify all operations and attributes related to individual items, such as *Print_Item*, *Delete_Item*, and so on.

2. The second step is to look for “natural homes” for these contract-based collections of functionality and then migrate them there. In this example, we gather operations related to catalogs and migrate them from the *LIBRARY* class and move them to the *CATALOG* class, as shown in Figure 5.5. We do the same with operations and attributes related to items, moving them to the *ITEM* class. This both simplifies the *LIBRARY* class and makes the *ITEM* and *CATALOG* classes more than simple encapsulated data tables. The result is a better object-oriented design.

3. The third step is to remove all “far-coupled” or redundant, indirect associations. In the example, the *ITEM* class is initially far-coupled to the *LIBRARY* class in that each item really belongs to a *CATALOG*, which in turn belongs to a *LIBRARY*.

4. Next, where appropriate, we migrate associates to derived classes to a common base class. In the example, once the far-coupling has been removed between the *LIBRARY* and *ITEM* classes, we need to migrate *ITEMs* to *CATALOGs*, as shown in Figure 5.6.

5. Finally, we remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments. In our example, a *Check_Out_Item* or a *Search_For_Item* would be a transient process, and could be moved into a separate transient class with local attributes that establish the specific location or search criteria for a specific instance of a check-out or search. This process is shown in Figure 5.7. ”

Specification 1: *Description of the strategy to remove a Blob. From [BMMIM98, page 77].*

are reproduced in this section for reference purposes², alongside their analysis.

Correction specification from Brown *et al.*

In the correction specification extracted from [BMMIM98, page 77] the general process of design smell correction can be observed (see Specification 1). The authors split the correction specification into several consecutive steps. Let us take the first two steps of the specification reproduced in Specification 1 as an example. These are focused on migrating cohesive sets of fields and methods from the *Blob* class to another class. Each of these steps involves a single type of refactoring, or a set of related refactorings, and presents a common procedure. In the first two steps, the objective is to apply **MOVE FIELD** and **MOVE METHOD**. First of all, the parameters of each transformation step have to be determined. In this case, the computation of parameters implies identifying which fields and methods constitute the cohesive sets and therefore, have to be moved away from the *Blob* class. The other important parameter of these

²The references cited within these specifications correspond to references in the original work. They have been kept in the excerpts in order to maintain the integrity of the original text. These cites do not correspond to references in this dissertation.

transformations is the target class to which the *Blob* class members are being moved. For each cohesive set, the specification compels the developer to find out what is the best place to move them to. The author does not describe how to proceed when an appropriate target class does not exist within the system, although in this case, we should probably want to create an empty class. The next part of the procedure is to apply the intended refactorings using the parameters previously identified. In the current example, **MOVE FIELD** / **METHOD** refactorings have to be applied for each cohesive set and target class, and for each field and method in the cohesive set.

The other detail worth mentioning from this specification is how the system is queried to find out the proper parameters for the proposed refactorings. In order to select the cohesive sets of fields and methods, the specification proposes to “*identify or categorize related attributes and operations according to contracts*”. The selection of the target class to which these sets should be moved would be done by searching for “*natural homes for these contract-based collections*”. These recommendations rely on the developer’s experience to manually instantiate the parameters of the refactoring specification. Nevertheless, the running example used in the specification gives an implicit hint: to look for similarities in the name of class members and group together those sets of members with similar names, and to determine the target class these sets should be moved to, on a basis of name similarity too. To summarise this, we can say that this correction specification compels the developer to instantiate and apply it by querying the system on a lexical basis and on the developer’s experience.

Correction specification from Fowler *et al.*

More lessons can be learned by analysing the correction specification, on how to remove a *Large Class*, extracted from [FBB⁺99, page 40] (see Specification 2). The proposed specification is based on the same principles: splitting the *Large Class* and redistributing its members to cohesive and single-responsibility classes. Additionally, it includes further details on what refactorings should be applied and in which circumstances. This specification explicitly suggests the refactorings to use, including **EXTRACT CLASS**, **EXTRACT SUBCLASS**, **EXTRACT METHOD**, etc. Another difference between this specification and the previous one is that the creation of new classes is explicitly stated through the **EXTRACT CLASS** / **SUBCLASS** refactorings.

The same “compute the refactoring parameters then apply the refactoring”-schema can be found in this specification, but an organized and ordered sequence of multiple steps is not shown. The specification describes different approaches or ideas, such as: **EXTRACT CLASS** / **SUBCLASS** / **INTERFACE** to split the *Large Class* and redistribute its responsibilities; **EXTRACT METHOD** to reduce the size of the class by removing duplicated code; **DUPLICATE OBSERVED DATA** to simplify a GUI class. These specifications can be tried in an unspecified order and can suit different cases of *Large Class*.

As for which type of queries this specification proposes, we can find references to lexical queries (“... *common prefixes or suffixes* ...”), search for fragments of duplicated code, structural queries (“... *how clients use the class* ...”), and identification of the kind of a class (“*If your large class is a GUI class* ...”). It can also be guessed that the assistance of the developer may be needed in most cases, since the tips given to discover the exact usage for each refactoring are rather subjective.

“ When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind.

You can *Extract Class* to bundle a number of the variables. Choose variables to go together in the component that makes sense for each. For example, "depositAmount" and "depositCurrency" are likely to belong together in a component. More generally, common prefixes or suffixes for some subset of the variables in a class suggest the opportunity for a component. If the component makes sense as a subclass, you'll find *Extract Subclass* often is easier.

Sometimes a class does not use all of its instance variables all of the time. If so, you may be able to *Extract Class* or *Extract Subclass* many times.

As with a class with too many instance variables, a class with too much code is prime breeding ground for duplicated code, chaos, and death. The simplest solution (have we mentioned that we like simple solutions?) is to eliminate redundancy in the class itself. If you have five hundred-line methods with lots of code in common, you may be able to turn them into five ten-line methods with another ten two-line methods extracted from the original.

As with a class with a huge wad of variables, the usual solution for a class with too much code is either to *Extract Class* or *Extract Subclass*. A useful trick is to determine how clients use the class and to use *Extract Interface* for each of these uses. That may give you ideas on how you can further break up the class.

If your large class is a GUI class, you may need to move data and behavior to a separate domain object. This may require keeping some duplicate data in both places and keeping the data in sync. *Duplicate Observed Data* suggests how to do this. In this case, especially if you are using older Abstract Windows Toolkit (AWT) components, you might follow this by removing the GUI class and replacing it with Swing components. ”

Specification 2: *Description of the Large Class smell and how to remove it. From [FBB⁺99, page 40].*

Correction specification from Wake

The specification to correct a *Large Class* from [Wak03, page 26] (See Specification 3) is very similar to that of [FBB⁺99, page 40]. Indeed, it is mainly a better structured version of it. However, it adds a tip on how to search for *Long Method* and tackle this smell first. Therefore, it implies the idea of composing correction procedures similar to those we have already found in other authors. The composition of correction specifications should be present in a framework aimed at supporting design smell correction in a comprehensive way.

Correction specification from Demeyer *et al.*

The correction specification for *God Class* from [DDN08, page 263] follows the same pattern (See 4), which is common to most of the analysed authors. It describes a multiple step process, where each step focuses on what refactorings to apply and which parameters to use. In this specification the authors reference, implicitly or explicitly: **EXTRACT CLASS**, **MOVE METHOD** (with delegation), **REMOVE MIDDLE MAN**, **REMOVE CLASS**. It does not reveal explicitly how to find out the refactoring parameters or how to decide on the exact refactorings to use. However, we can spot the need for structural queries, metrics and, if none of these help, the developer's intuition. Relationships with other design problems can also be recognized. In addition to *God Class*, this specification involves splitting this class into what are likely to be *Data Classes*. A restructuring pattern named **MOVE BEHAVIOR CLOSE TO DATA** is recommended in order to move the corresponding methods left behind. Therefore, this suggests addressing the reorganization of these methods as if they would be affected of *Feature Envy*, and indeed, they probably

“ In general, you’re trying to break up the class. If the class has Long Methods, address that smell first. To break up the class, three approaches are most common:

- *Extract Class, if you can identify a new class that has part of this class’s responsibilities*
- *Extract Subclass, if you can divide responsibilities between the class and a new subclass*
- *Extract Interface, if you can identify subsets of features that clients use*

Sometimes, the class is big because it’s a GUI class, and it represents not only the display component, but the model as well. In this case, you can use Duplicate Observed Data to help extract a domain class. ”

Specification 3: *Description of the strategy to remove a Large Class. From [Wak03, page 26].*

will, once the previous redistribution of attributes is performed.

Correction specification from Lanza and Marinescu

The correction specification for *God Class* in [LM06, page 83] is less detailed than the previous ones in terms of which refactorings to apply and how to find the parameters for them (see Specification 5). The basic correction idea is vaguely mentioned in this specification: to redistribute the cohesive members from the *God Class* to other classes. They do not describe how to remedy this smell in detail, but instead they refer to other authors’ specifications. Nevertheless, it explicitly mentions a very important consideration. This smell (or “disharmony”, according to them) may have been formed from a confluence of others. Indeed, this concept of composition and relationship between smells is studied in detail in [LM06]. As for developing an automated approach for design smell correction, this is a very relevant notion. This implies that correction methods can be composed from others, therefore allowing for reuse and incremental evolution and improvement of correction specifications.

Correction specification from Kerievsky

In [Ker04], Kerievsky compiles a catalogue of design-pattern-related refactorings that are matched against design smells, and can therefore be used to correct them. The correction specifications from [Ker04, page 44] are described by means of particular structures that the system has to be migrated to. Before trying to apply a correction procedure, the developer has to identify which particular incarnation of *Large Class* the system presents. To remove a *Large Class*, three different correction specifications are proposed (see Specification 6). Depending on the kind of problem, the system will have to be migrated to a different structure, and the developer should use the particular correction specification that fits.

These design-pattern-related refactorings, describing how to migrate the code to the desired structure, are detailed in [Ker04], and are defined in the style of the refactorings specifications defined in [FBB⁺99]. Reviewing them (see [Ker04, page 166, page 192, page 269]), reveals that each one is defined like a “big refactoring”[BF99b, chapter 12], and therefore, they are composed from other refactorings and smell correction specifications.

The relevant idea that can be observed in this specification is that the approach to correct a design smell can be selected from several different methods. This implies that the knowledge for correcting a design smell can grow over time, and be compiled in catalogues, as we are able to

“Incrementally redistribute the responsibilities of the god class either to its collaborating classes or to new classes that are pulled out the god class. When there is nothing left of the god class but a facade, remove or deprecate the facade. . . . The solution relies on incrementally moving behavior away from the god class. During this process, data containers will become more object-like by acquiring the functionality that the god class was performing on their data. Some new classes will also be extracted from the god class.

The following steps describe how this process ideally works. Note, however, that god classes can vary greatly in terms of their internal structure, so different techniques may be used to implement the transformation steps. Furthermore, it should be clear that a god class cannot be cured in one shot, so a safe way to proceed is to first transform a god class into a lightweight god class, then into a Facade [p.319] that delegates behavior to its acquaintances. Finally, clients are redirected to the refactored data containers and the other new objects, and the Facade can be removed. The process is illustrated in Figure 39.

The following steps are applied iteratively. Be sure to apply Regression Test After Every Change [p. 201]:

- 1. Identify cohesive subsets of instance variables of the god class, and convert them to external data containers. Change the initialization methods of the god class to refer to instances of the new data containers.*
- 2. Identify all classes used as data containers by the god class (including those created in step 1) and apply Move Behavior Close to Data to promote the data containers into service providers. The original methods of the god class will simply delegate behavior to the moved methods.*
- 3. After iteratively applying steps 1 and 2, there will be nothing left of the god class except a facade with a big initialization method. Shift the responsibility for initialization to a separate class, so only a pure facade is left. Iteratively redirect clients to the objects for which the former god class is now a facade, and either deprecate the facade (see Deprecate Obsolete Interfaces [p. 215]), or simply remove it.*

”

Specification 4: *Description of the strategy to remove (split up) a God Class. From [DDN08, page 263].*

identify more precisely the specific manifestations of a smell and their corresponding correction procedures. It is also worth noticing that this defines an obvious but important opportunity for automation. We can group these different methods and let an automated tool select the proper one for each case if we can provide a way to identify each particular manifestation of the problem. Queries over the system structure, metrics, and developer assistance can be employed to make this decision.

Correction specification from Trifu

The most detailed specifications can be found in [Tri08, page 32,39]. Trifu compiles very detailed algorithmic descriptions that use `if-then-else`, `foreach` constructs, etc. They resemble the specifications of Kerievsky in the sense that we need to identify the specific incarnation of the smell first. What Trifu calls the “reference structure” is the structure that realizes the original design intent. The detailed description leads to the transformation of the smelly part of the system into this desired structure. This work is very relevant because it reveals it is possible to write detailed specifications, that can be easily automated, when the precise manifestation of a design smell has been identified.

The proposed algorithm includes, either implicitly or explicitly, applying refactorings from [Opd92]: **CREATE EMPTY CLASS**; from [FBB⁺99] such as: **ENCAPSULATE FIELD**, **MOVE**

“Refactoring a God Class is a complex task, as this disharmony is often a cumulative effect of other disharmonies that occur at the method level. Therefore, performing such a refactoring requires additional and more fine-grained information about the methods of the class, and sometimes even about its inheritance context. A first approach is to identify clusters of methods and attributes that are tied together and to extract these islands into separate classes. Split Up God Class [DDN02] proposes to incrementally redistribute the responsibilities of the God Class either to its collaborating classes or to new classes that are pulled out of the God Class. Feathers [Fea05] presents some techniques such as Sprout Method, Sprout Class, Wrap Method to be able to test legacy system that can be used to support the refactoring of God Classes.”

Specification 5: *Description of the strategy to remove a God Class. From [LM06, page 83].*

“Fowler and Beck [F] note the presence of too many instance variables usually indicates that a class is trying to do too much. In general, large classes typically contain too many responsibilities. Extract Class [F] and Extract Subclass [F], which are some of the main refactorings used to address this smell, help move responsibilities to other classes. The pattern-directed refactorings in this book make use of these refactorings to reduce the size of classes.

Replace Conditional Dispatcher with Command (191) extracts behaviour into Command [DP] classes, which can greatly reduce the size of a class that performs a variety of behaviours in response to different requests.

Replace State-tering conditionals with State (166) can reduce a large class filled with state transition code into a small class that delegates to a family of State [DP] classes.

Replace Implicit Language with Interpreter (269) can reduce a large class into a small one by transforming copious code for emulating a language into a small interpreter.”

Specification 6: *General Description of three strategies to remove a Large Class. From [Ker04, page 44].*

FIELD, MOVE METHOD, EXTRACT METHOD, PUSH DOWN FIELD, PUSH DOWN METHOD, EXTRACT SUBCLASS; and from [Ker04]: **IMPLEMENT “FACADE”**.

To find out how to refactor, the specification instructs the developer on how to examine the situation at each step. In some cases, deciding on the refactoring parameters only seems only possible with the developer’s deduction (*“Decide between keeping the structured type ... or increasing the association’s multiplicity ...”*), but other directions can definitely be translated to structural queries (*“if m_i is specialized in one of O ’s subclasses then”*), and metrics (*“Identify all the abstractions A_i that need to be separated...”*).

It should be noted that different authors can propose different correction procedures. For example, Specification 2 is similar to Specification 1, but it includes the use of extract subclass, extract interface, extract method, duplicate observed data, replace algorithm, etc. The specifications from [Tri08, page 32,39] can coexist with those from [Ker04] because they apply to different manifestations of the same problem. It would be difficult to integrate all the correction proposals into a single correction specification. Some specifications can complement or refine others, while others cannot. Different specifications can be applied to suit the different types of manifestations of the smell. This has been quite rightfully identified by Trifu in [Tri08]. A correction approach has to take into account these considerations.

```

1: Let O be the schizophrenic class
2: Check that we have an identity
   based decomposition of data in O, based
   on the identities of the encapsulated ab-
   stractions. // An action oriented topol-
   ogy in the case of the encapsulated abstrac-
   tions would require a complete redesign of
   the fragment, which is outside the scope of
   design flaws in general (see 6.1.1)
3: Encapsulate all attributes in O with
   public accessors // The public visibility is
   only temporary, in order to make moving
   functionality around easier
4: Identify all the abstractions Ai ,
   that need to be separated and establish
   their future interfaces
5: Create empty classes that corre-
   spond to each of Ai
6: if O has subtypes // we assume that
   they contain valid specializations for one
   or more of the abstractions contained in
   O then
7: Establish those abstractions from Ai
   that are affected by each specialization of
   O
8: Create appropriate subtypes for the
   classes that correspond to these abstrac-
   tions
9: end if
10: for all attributes ai in O do
11: Find the natural place for ai in
   one of the newly created classes, including
   helpers, based on the Ai determined before
   and apply "move field" [Fow99]
12: if ai is an array or collection then
13: Decide between keeping the struc-
   tured type and having an association mul-
   tiplicity of 1 to the host class, or increas-
   ing the association's multiplicity and re-
   placing the collection or array with only
   one of its elements. In the latter case, the
   class interface and the implementation of
   the facade need to be adapted accordingly
14: end if
15: end for
16: for all methods mi in O do
17: if mi can be unambiguously as-
   signed to one of the new classes then
18: Apply "move method" [Fow99] to
   move mi's body to the respective class
19: if mi is specialized in one of O's
   subclasses then
20: Apply "move method" [Fow99]
   to move the overriding method into the
   appropriate specialization
21: end if
22: else
23: Apply "extract method" and "move
   method" [Fow99] to break up the origi-
   nal method, based on the attribute clusters
   that determine the encapsulated abstrac-
   tions, and reunite functionality with its
   associated data
24: if mi was previously specialized in
   one of O's subclasses then
25: Apply "extract method" and "move
   method" [Fow99] to break up the original
   overriding method, based on the attribute
   clusters that determine specializa-
   tions of the encapsulated abstractions, and reunite
   functionality with its asso- ciated data
26: end if
27: end if
28: if mi had public visibility then
29: Implement "facade" [GHJV96]
   method in O, delegating to the appropri-
   ate abstraction(s).
30: end if
31: end for
32: Create initialization methods in
   the facade O, or adapt its constructors to
   instantiate and wire together all newly de-
   fined classes and their specializations
33: Reduce data and accessor visibil-
   ity as much as possible in all of the newly
   created classes "

```

Specification 7: Description of the strategy to remove a Schizophrenic Class. From [Tri08, page 32,39].

Other related works

There are other works related to removing design smells that do not consist of catalogues of correction specifications, but describe techniques that support the restructuring process in some way. All the correction specifications analysed describe the same need when dealing with a *Large Class*: to identify the optimal cohesive sets of fields and methods to be move and how to find their best new locations.

Some authors propose techniques based on metrics to reorganise a system’s design and to migrate it to structures which can be optimal for a given measure. These approaches usually produce refactoring suggestions, but do not offer or allow for an automatic way to apply them. Bodhuin *et al.* introduced SORMASA in [BCT07]. They presented a tool that can suggest the refactorings needed to achieve a desired system structure. Their tool uses genetic algorithms to propose alternative structures and to select the one that optimizes a given fitness function.

In her PhD Thesis dissertation [Moh08], Moha proposes a method, based on Relational Concept Analysis (RCA), to obtain refactoring suggestions. Some design problems, such as “The Blob” [BMMIM98], can be reduced by reorganising and redistributing classes and their features with refactorings. RCA helps to discover how to redistribute responsibilities between classes so coupling and cohesion metrics are improved. The computed redistribution leads to the refactorings that should be used to remove the design smell.

Some authors have developed more systematic approaches that make it easier to automate the correction of design smells.

In [TSG04], although they do not explicitly present correction specifications, Trifu *et al.* presented a technique to detect and remove design smells. Correction is based on the use of what they call “correction strategies”. These strategies are specifications for design smell correction procedures that are written in terms of elaborated algorithms. They are composed of code transformation rules –refactorings and non-behaviour preserving transformations–, and conditional and iterative constructs. These strategies are written in INSPECT/J [GKB07] and represent alternative branches of simple transformations. A particular path is walked depending on how it favours or disfavours certain quality factors. If a path cannot be followed because of the precondition of a transformation does not hold, another path, or user intervention, is considered. Simple transformations may include refactorings and non-behaviour preserving transformations, but a correction strategy, as a whole, does preserve behaviour.

In [TK04], Tahvildari *et al.* present an approach that uses quality models. The impact of a set of refactorings over metrics and quality factors is described with soft-goal models [CNYM99]. These models are then used to automatically select the best refactorings for a given goal, such as improving maintainability. They compute the potential refactorings which are appropriate to remedy the desired problem. Using refactoring preconditions, they filter the non-applicable transformations out of the set of candidate refactorings. Nevertheless, they do not specify in detail how to obtain the precise sequence of refactorings that should be applied. We find this to be a limitation. Refactorings which are not straightforwardly applicable are discarded, so their approach may perhaps produce sub-optimal solutions. Their approach can possibly be improved by allowing it to select the optimal refactorings, regardless of their immediate applicability.

4.1.2 A model for current design smell correction specifications

We have reviewed a set of works related to design smell correction with refactorings. From this analysis we have extracted a common general model, which is shown in Figure 4.1, that can

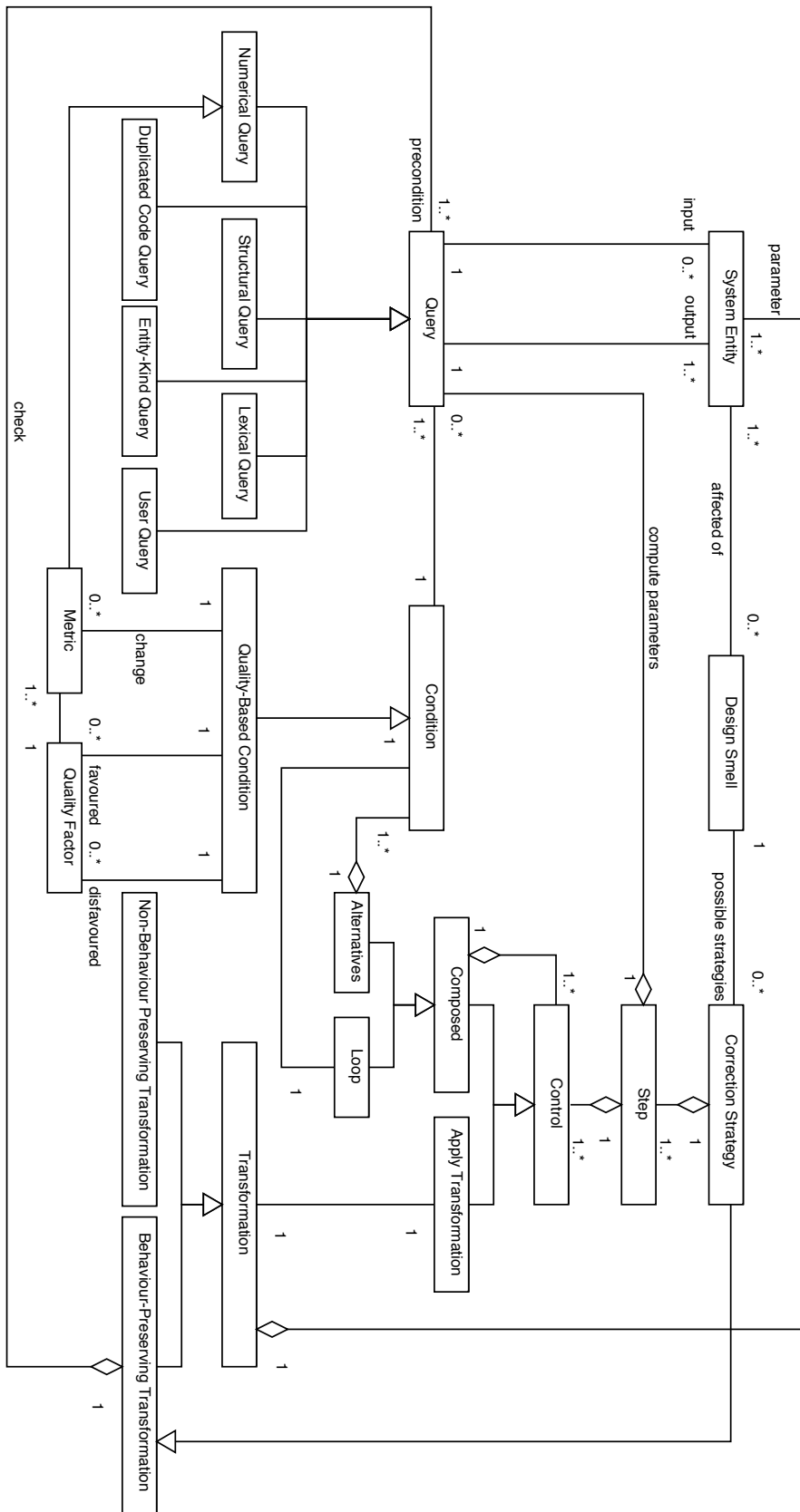


Figure 4.1: *A general model for current design smell correction strategies.*

represent all those different approaches.

This model represents the different kinds of design-smell-removal specifications as *correction strategies*. This is meant to be a unifier concept, and its name has been taken from [TSG04]. A *design smell* can affect a number of *system entities*. This relationship denotes only those entities which are strictly “affected by” the smell. For example, a method accessing too many member variables from another class is affected by *Feature Envy* and is represented by this relationship, but not the envied entities, despite being involved in the smell. *Correction strategies* constitute the different “possible strategies” that can be employed to remove a design smell. These strategies are defined to alter only the structure of a system, not its behaviour, therefore, they are a particular kind of *behaviour-preserving transformation*.

Correction strategies are composed of sequential, ordered *steps*. They can thus be performed manually in an easier way. Each step instructs the developer to apply some transformations following a certain algorithm. Prior to performing the corresponding transformations, their parameters have to be computed using *queries* that gather different types of information from the system. The algorithm comprising each correction step can be formulated by the composition of different *control* constructs. This is represented by specialising each construct, either into a *composite* control construct or into a simple precept, to *apply a transformation*. *Composite* constructs can be *alternatives* or *loops*, both of which include the evaluation of a *condition*.

In short, the basic set of pseudo-code constructs which have been found in the existing correction specifications are:

- ***Applying a transformation:*** which is the basic operational construct. This will include applying any type of transformation, such as refactoring, introducing or removing a design pattern [Ker04, Tri08], or even performing non-behaviour-preserving transformations, as parts of a behaviour-preserving one.
- ***Alternatives:*** to specify different transformation paths that will be selected upon the evaluation of a condition.
- ***Loops:*** to specify iterative transformations that have to be applied on the evaluation of a condition. We have found that the majority of loops are “foreach” loops, used to apply a transformation iteratively over a set of system entities.
- ***Conditions:*** that can be evaluated over system queries.

We have found that the *transformations* that can be recommended to apply in the correction process can be either *non-behaviour preserving* or *behaviour preserving*. Both transformations should define a certain algorithm which implements the associated changes, but the latter can be seen as a conditional transformation [Kni06], which is also composed of a “precondition”. This precondition is evaluated upon the execution of certain system queries. The fulfillment of this precondition is a requirement for the application of a behaviour-preserving transformation, because it denotes the condition under which the application of the transformation does not alter the observable behaviour of the system. Both types of transformations, which can be included in a correction specification, can be represented as reusable, parameterised transformations which take *system entities* as arguments –including classes, methods, etc.– which are affected by a particular smell or those which are involved in the correction process.

The availability of *system queries* is crucial in correction strategies. They can be used to gather additional information during the correction process. System queries can be formulated

so that they use *system entities* as input parameters or as output results. The different types of system queries that we have been identified are:

- **Structural**: finding those system entities which present a particular structural pattern, which can range from simple AST relationships to complex structures, such as the “pathological structures” used in [Tri08].
- **Numerical**: extracting numerical data, such as metrics, from system entities [LM06].
- **Lexical**: certain name patterns, prefixes and suffixes, such as: “System”, “Process”, etc. can reveal entities that should be involved in the correction process [Moh08].
- **Duplicated code**: Some transformations would require the ability to identify duplicated code. For example, the refactoring **EXTRACT METHOD**, that is used to unify duplicated code fragments from different methods into a single method, could be better automated with the help of a query that detects and finds duplicated code.
- **Entity-kind**: The kind of an entity – *i.e.* a class can be a GUI class, a library class, etc. – can determine which is the most suitable strategy to correct a smell, or even whether the entity is affected by a smell or not (Specifications 2, 3).
- **User**: The user of a correction strategy – the system developer – can use his or her expertise in the correction process or take decisions which are hard to automate. We have represented this as a *user query* to highlight the idea that human interaction can be employed in an automated approach. This can be implemented as a consult the system would make to the developer requesting additional information.

We have found that some approaches describe correction specifications that can also include decision points and alternative paths which rely on *quality-based conditions*. This kind of conditions are used to describe the effect of a transformation over a *metric* or a *quality factor* [TK04, TSG04]. These can be used to choose between the different candidate strategies that may possibly be used to correct a smell or to choose between different transformation paths within a correction strategy.

4.1.3 Current specifications of refactorings

In order to integrate design smell correction specifications with refactoring specifications, a model for the latter is also presented here. The information for this model has been extracted by analysing how different authors have defined refactorings in the literature [Opd92, FBB⁺99, Tic01, MC03, Ker04, Kni06]. Figure 4.2 shows this simplified model for refactoring operations.

A *refactoring* definition includes a list of parameters, a precondition and an algorithmic description of the refactoring mechanics. The parameters of a refactoring will be suited by different types of *program elements*. The actual parameters of a refactoring will determine the precise program elements over which the refactoring will be applied, or those elements that will be involved in the transformation process. The precondition of a refactoring will be defined with a *boolean condition*, which can be evaluated into “true” or “false” at application time. As it is already well known, the fulfillment of the precondition ensures, with a high degree of certainty, that the application of the refactoring will not alter the observable behaviour of a program. A refactoring will only be applied when the precondition evaluates to “true”. The

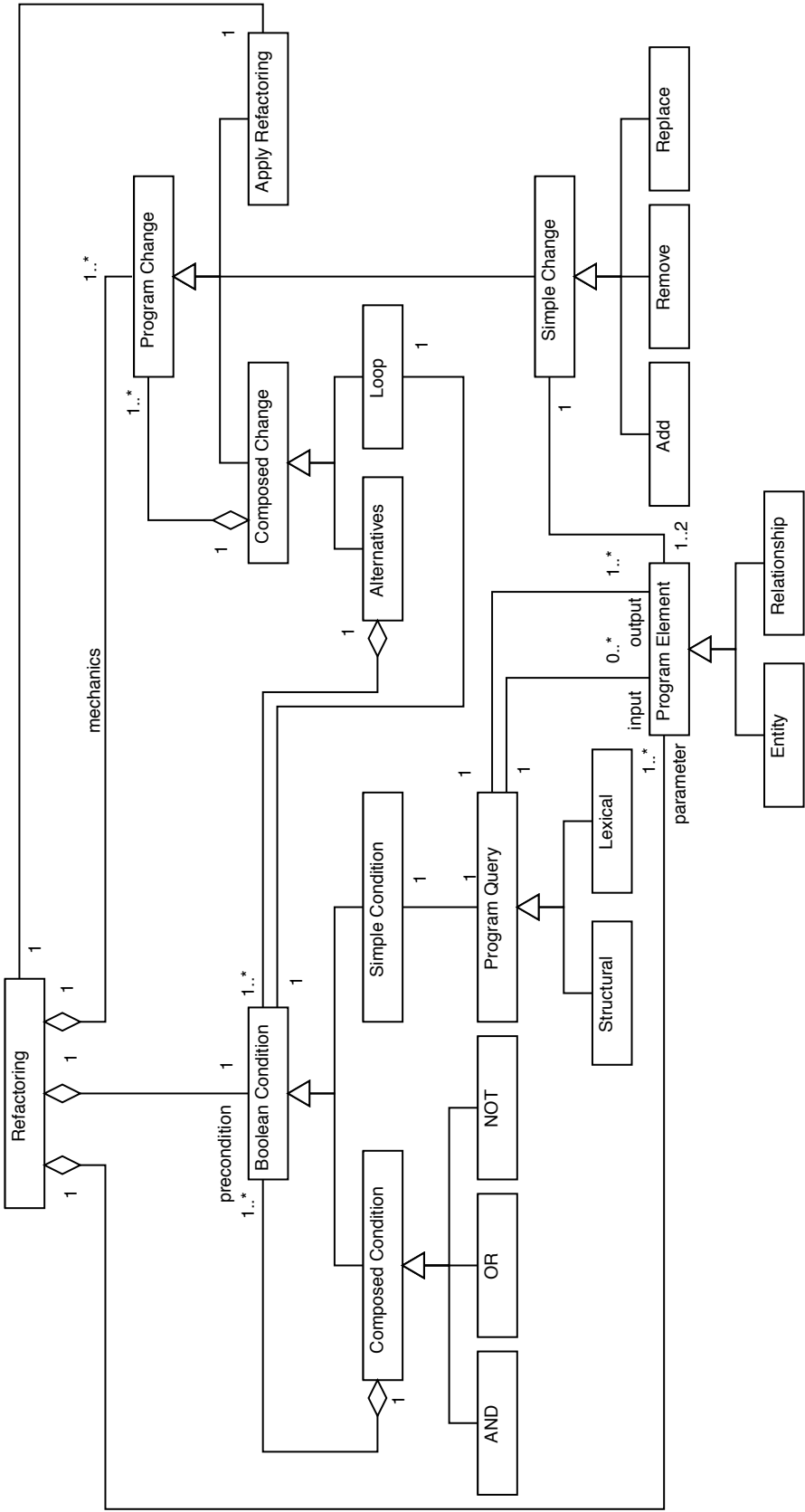


Figure 4.2: A simplified model for refactoring operations.

mechanics of the refactoring describes the necessary steps or *program changes* to apply the desired transformation. These are described with different levels of detail and usually, in a heuristic way.

A *boolean condition* is built from *composite* or *simple conditions*. The former are composed by joining boolean conditions with boolean operators, while *simple conditions* are formulated as expressions over *program queries*. The program to be transformed can be seen as a collection of *program elements* which build up a program model. Refactoring specifications, which are manually-oriented, refer to source code entities, while automated refactoring tools can rely on different types of program models, such as Abstract Syntax Tree (AST) elements, JAVA bytecode, logic formulas, etc. In order to abstract these options, *program elements* are represented within the refactoring model in Figure 4.2, in a generic way, by specialising them into *entities* and *relationships* between entities. All the computations, explorations, etc. needed to obtain additional information about the program have been represented as *program queries*, which can either be *structural* or *lexical*. These queries have *program elements* as their input parameters and output results.

The mechanics of a refactoring are defined, in most cases, as a sequence of *program changes*, which can be specialised into a *simple change*, a request to *apply a refactoring* or into a *composite change*. A *composite change* can be represented in a simplified way by *alternatives* and *loops*, while a *simple change* is a basic request to either *add*, *remove* or *replace* a *program element*.

4.1.4 The activities of applying a design smell correction strategy

To fully understand the problem of performing complex refactoring processes, we summarise here the general process of applying a design smell correction strategy. In order to remove a design smell, a correction strategy has to be selected. Moreover, we also need to plan ahead how to instantiate that correction pattern for each specific situation. This involves, for each step in the correction strategy, computing the precise parameters and finding out how to apply the transformations included in the step. This can be seen as a three-stage process:

1. Select the most adequate strategy to correct a smell.
 - We should determine which sequence of refactorings can be used to remove a smell: *eg.* if a certain method suffers from “Feature Envy” [FBB⁺99, page 80], in order to remove this smell, the method should be moved close to the data it accesses, from its source class to the class containing most of the accessed data. This stage involves deciding which strategies are suitable to be applied. In some cases, this can be guided by additional information. For example, identifying which particular incarnation of a design smell we are facing may help.
2. Instantiate the selected strategy.
 - We should have to find the precise strategy. For each step in the correction strategy, the intended transformation sequence has to be parameterised with the entities of the system over which it will be applied. Following the previous example, the strategy would be instantiated for a method that suffers from “Feature envy”, its source class and a target-candidate class.

3. Apply the strategy instance.

- We should have to find the exact transformation sequence, including refactorings and non-behaviour-preserving transformations, needed to enable the application of the desired instance of the correction strategy. In the example, if the method cannot be moved due to a name conflict, an additional or alternative sequence of refactorings should be proposed. For example, prior to moving the method, a renaming of one of the conflicting methods could also be applied.

4.2 Open problems in automating correction strategies

The models in Section 4.1, together with the survey in Chapter 3, describe the current state of the art in design smell correction automation as far as our knowledge allows. Through our analysis, we have detected some problems and issues that have not been studied with enough attention.

4.2.1 Applicability of refactorings

One of the problems that have not been addressed in detail is the applicability of the transformations involved in the correction process. As mentioned in Section 2.3, checking a refactoring precondition must precede its actual execution. If these conditions, that guarantee the preservation of the system's behaviour, do not hold, then it is not safe to apply the refactoring, and therefore, a refactoring tool will not execute the transformation. For example, to allow a method to be moved from one class to another, one should probably, first, have to move the attributes accessed from it. In these cases, the desired refactoring can not be immediately applied and the situation must be analysed in order to find how to apply the desired change. Either an alternative refactoring sequence is selected, or some preparatory refactorings are executed first.

To make this problem worse, the dependencies and conflicts between refactorings have to be taken into account. Two refactorings are in conflict if one of them can disable the precondition of the other. On the contrary, they are dependent when one can enable the precondition of the other. This problem, regarding conflicts and dependencies between refactorings, can appear in different development contexts, and has been studied by some authors. Graph transformation has been used as the formal-basis to analyse the dependencies and conflicts between refactorings by some authors, such as [MTR07, PRT08]. Logic programming has also been tested as a suitable formalism to deal with this problem [MKR06].

Team development is one of the most common scenarios where this problem exists. If two developers are working on the same system at the same time, performing refactorings and edits in parallel will produce conflicts when merging the different system versions, even if the merge is performed with the aid of a software configuration management system. This particular scenario is addressed in [DMJN08]. In this work, Dig *et al.* describe how the dependencies between refactorings have to be analysed and the composite sequence of changes has to be reorganised in order to produce a correct merging. Preconditions are used to determine the valid refactoring commutations.

These precondition-related problems do not only apply to simple or low-level refactorings, but to all behaviour-preserving transformations which are “protected” with a precondition. These include the kind of transformations we are addressing in this dissertation: big refactorings such as introducing or removing a design pattern, removing a design smell, etc. Therefore, in order

to support more automated correction strategies, we need an infrastructure “smart” enough to compute alternative transformation paths and to find a proper composite sequence of refactorings that does not contain “failing” preconditions, and thus, can be immediately executed over the current system.

4.2.2 Non-formal description of correction strategies

Another source of concern arises from how the strategies are specified. Most strategies we have found can be formulated in an algorithmic way. Obviously, any algorithmic language will be expressive enough to describe them. Many specifications can be directly translated to pseudo-code. However, they are often written in natural language and are also given in a heuristic way. This makes it more difficult to translate the strategy specifications to an algorithmic language, or at least, to a deterministic one. In order to write down these specifications, some non-deterministic language constructs are needed.

In addition to the regular control constructs we have identified in correction strategies (see Figure 4.1), it will also be desirable to extend them with non-deterministic features:

- **steps:** Executing a sequence of operations requires all steps in the sequence to be applied in order and successfully for the sequence to succeed. Due to the heuristic nature of correction strategies, it is desirable to have the chance to specify operations in a less restricted way. It would be useful to allow for non-deterministic steps that may be carried out optionally. This refers to steps which are worth trying but are not needed for the whole strategy to succeed. It would also be useful to allow scheduling steps in an unordered way, so that all steps should be executed but no order is explicitly specified.
- **alternatives:** When specifying alternatives, these are given a predefined order: an alternative will be tried only if another one fails. Instead of “hard-coding” this, for a heuristic correction strategy it would often be preferable to give the same priority to a set of alternatives and let a tool select and execute the proper one. Additionally, regular alternative constructs are usually controlled by some kind of condition, as in an “if-then-else” construct. An approach for writing and supporting automation of correction strategies should allow to omit the condition of an alternative branch. This will allow the developer to specify an alternative step or strategy even if the condition to take that path is not fully known or cannot be described for some reason.
- **foreach loops:** When iterating over a set of entities to perform a certain transformation, the success of the transformation as a whole may depend on the order in which each element in the set is selected to execute the loop body over this selection. An example of this could be removing, in individual steps, a set of methods which are only referenced from methods belonging to that set (see **DELETE MEMBER FUNCTIONS** in [Opd92, page 59]). As long as there are no cycle dependencies between them, of course, the first methods to be removed would be those which are completely unreferenced. Other orderings shall fail. As is the case with *unordered alternatives*, it is convenient to leave the ordering unspecified in the correction strategy, while an automated tool, which is able to compute the successful ordering, is then needed.

Additionally, we can identify some instructions within correction strategies that should be supported by an automated tool but currently are not.

- ***apply a strategy / substrategy***: Instead of a direct invocation of a refactoring, which could fail due to its associated precondition, it would be desirable to support the invocation of another correction strategy or substrategy. This will allow parts of strategies to be split and reused.
- ***calls for user interaction***: this is highly desirable since it will allow the application of a correction strategy to be guided by the user when the strategy is not “smart enough” to be applied in a fully automated way.

Once we have modelled the current state of the art in design smell correction automation, and given that we have also identified its open problems, we propose a framework that we consider can lead to an improvement in this field.

4.3 Definition of refactoring strategies

In order to tackle the problems mentioned in Section 4.2 we propose to generalise the design smell correction approaches into a wider approach. ***Refactoring Strategies*** unify the analysed approaches and present a framework in which all tasks involved in the design smell correction process are given the same degree of attention.

4.3.1 Overview of refactoring strategies

We present ***refactoring strategies*** as specifications, that can be easily automated, of the steps to perform complex behaviour preserving transformations. A refactoring strategy compiles the empirical knowledge on how to achieve a particular ***goal*** that needs to be reached by application of complex behaviour-preserving transformations, such as removing a design smell, introducing or removing a design pattern, applying a particular instance of a complex refactoring operation, etc. When intended for removing a design smell, a refactoring strategy represents a correction strategy, which is the problem that motivates and drives this PhD Thesis dissertation. Since refactoring strategies represent a unifying and homogeneous concept, they can be used in a uniform way to describe heuristics for any of those goals. They can also be automated in a uniform way in order to compute the sequences that achieve them. Refactoring strategies can be seen as a revised version of the correction strategies defined by Trifu [TSG04]. The added values of refactoring strategies are introduced below.

Refactoring strategies are instantiated into ***Refactoring Plans***, which are refactoring sequences that can be executed over the current system’s source code. Refactoring strategies and refactoring plans separate the specification of a correction strategy from a particular executable instance of that strategy, which is applicable over a system in a certain state. They allow a complex behaviour-preserving transformation to be planned ahead, so the sequence of instantiated refactorings included in the refactoring plan can be executed safely. The computation of refactoring plans from refactoring strategies forces the preconditions of all the refactorings in the resulting sequence to be fulfilled at the time of their application.

Refactoring strategies compile the empirically-obtained and experience-based knowledge on how to perform a complex refactoring process. When a certain process can be formulated in a precise way, the suitable strategy will be more detailed and the matching refactoring plan will be obtained in a more straightforward way by a tool. If not enough knowledge has been collected to address a refactoring process, a more vague specification can be written. Nevertheless, the

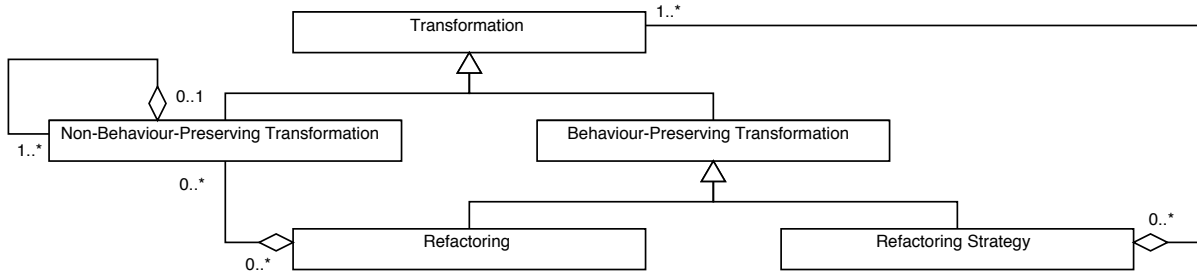


Figure 4.3: Refactoring strategies and their relationship with refactorings and non-behaviour-preserving transformations.

structured nature of refactoring strategies makes it easier for a tool to still be able to perform the necessary computations to find out the suitable refactoring plans.

The following definitions are proposed:

Definition 2. A **Refactoring Strategy** is a heuristic-based, automation-suitable specification of a complex behaviour-preserving software transformation which is aimed at a certain goal. It can be instantiated, for each particular case, into a *Refactoring Plan*.

Definition 3. A **Refactoring Plan** is a sequence of instantiated transformations, aimed at achieving a certain goal, that can be effectively applied over a software system in its current state, while preserving its observable behaviour. It can be an instance of a refactoring strategy.

The following sections define and describe the model we propose for refactoring strategies. To start presenting this proposal we introduce here an overview at the highest abstraction level. In our model, we propose to use three different types of software transformations, **Non-behaviour-preserving transformations**, **refactorings**, and **refactoring strategies**. A general model depicting the relationships between these transformations is shown in Figure 4.3 and detailed below:

- **Transformation:** A general kind of software system transformation, which can be specialised into *non-behaviour-preserving* and *behaviour-preserving transformations*.
- **Non-behaviour-preserving transformation:** Specifications of software transformations that may change the observable behaviour of the system. We propose that, in the case of being composed by other transformations, these can only be non-behaviour-preserving transformations. We thus enforce this kind of transformations to be simple and to serve as building blocks for refactorings and refactoring strategies.
- **Behaviour-preserving transformation:** Specifications of software transformations that do not change the observable behaviour of the system. In our proposal, they can be either refactorings or refactoring strategies.
- **Refactorings:** In our proposal, they are meant to refer only to low-level and simple refactorings. That is, refactorings that are defined by simple non-behaviour-preserving

transformations and never by other refactorings. Those refactorings that are composed of other ones, such as high-level and big refactorings, constitute complex refactorings and therefore should be specified with *refactoring strategies*

- **Refactoring strategies:** Specifications of complex refactoring processes, that are composed by transformations of any kind. They represent the concept described in Definition 2.

The next sections provide an in-depth description of our proposal for refactoring strategies. In order to do this, we present the proposal as two models, and we describe their elements and relationships.

4.3.2 A model for refactorings and non-behaviour-preserving transformations

In our proposal, the basic building blocks for refactoring strategies are low-level refactorings and non-behaviour preserving transformations. The main difference between refactoring strategies on the one side and refactorings and non-behaviour-preserving transformations on the other, is that, in our proposal, definitions for refactorings and non-behaviour-preserving transformations, are not based on heuristics, and therefore their execution is deterministic. The precise transformation sequence to be performed over the software system is determined by the refactoring or the non-behaviour-preserving transformation and their parameters. Once the transformation has been selected and the parameters have been established, the precise transformation sequence is already known. These transformations are defined by the model shown in Figure 4.4 and their elements are described below.

- **Non-behaviour-preserving transformation:** Specifications of deterministic algorithms to modify a software system. These transformations can be initiated within a *refactoring* or within another *non-behaviour-preserving transformation*. They take *system entities* as parameters and are composed of several *transformation steps* which constitute the “transformation algorithm”.
- **Transformation step:** Each transformation step of a *refactoring* or a *non-behaviour-preserving transformation*. It is specialised into three different kinds of steps: *apply transformation*, *system change* or *composite change*. Transformation steps should not be confused with those steps identified in current correction strategies (see Section 4.1.2). Transformation steps are atomic operations, while the latter are more coarse-grained and were used to reference sub-parts of a correction strategy.
- **Apply transformation:** A request to execute a *non-behaviour-preserving transformation*.
- **System change:** This represents a simple change that can be performed over the software system. It can be specialised into three different types of modifications: *add*, *remove* or *replace* a *system element*.
- **Composite change:** A modification to the system which is composed of other changes, each being any type of *transformation step*. The execution of the component changes can be structured by specialising a *composite change* into a *loop* or into *alternatives*.

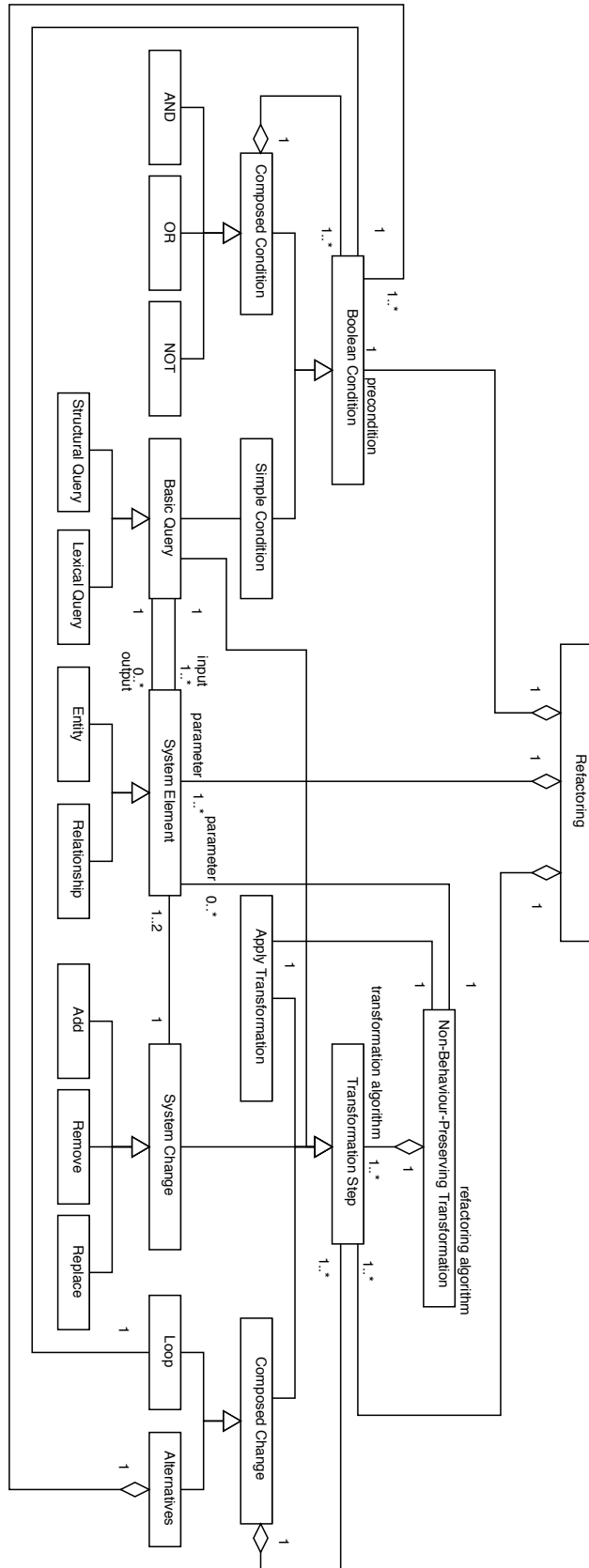


Figure 4.4: A model for the refactorings and the non-behaviour-preserving transformations used in refactoring strategies.

- **Loop:** A kind of *composite change* to specify that a set of transformations should be executed iteratively under the evaluation of a *boolean condition*. Different types of loops may be used being “while” loops the most basic ones. Special types of loops, used very often, are “foreach loops” for which the iterations are carried out over each element of a set of elements. In regards of this kind of loops, it should be useful to allow to iterate over a given or explicit collection of elements, or even to allow this collection to be created dynamically. This is to iterate over the elements which fulfill a given condition.
- **Alternatives:** A kind of *composite change*, that contain different sequences of transformations, guarded by *boolean conditions*. The different types of alternatives include simple alternatives – *if-then* and *if-then-else* alternatives – or multiple choice alternatives – *switch-case-case-...* alternatives .
- **System Element:** The lowest-level elements of the system that can be accessed, to manipulate them or to gather information about them. They can represent either *entities* or *relationships* between elements. The precise kinds of entities and relationships that an approach can deal with depends on the model of the system that the approach uses.
- **Boolean condition:** An expression that can be evaluated to *True* or *False*. It can take the form of *simple conditions* or *composite conditions*, which are built by joining together other *boolean conditions* with boolean operators. A *simple condition* is formulated as an assertion over a current property of the system, which is computed with a *basic query*.
- **Basic query:** A function that evaluates, searches or explores the model of the system to deduce certain information from it. These functions use *system elements* as their input parameters and as their output results. The different types of basic queries available are: *structural queries*, which are used to gather *system elements* that hold a particular structural property, and *lexical queries* which operate over the “names” or identifiers of the System Elements.
- **Refactoring:** A behaviour-preserving system transformation. It is composed of a *boolean condition* precondition, a set of parameters and a sequence of transformations steps.

4.3.3 A model for refactoring strategies

The analysis of the state of the art in design smell management presented in Chapter 3 and the domain analysis performed in this chapter, lead us finally to the definition of refactoring strategies, which we present as one of the major contributions of this dissertation. The specification of refactoring strategies, introduced previously in Definition 2, is completed with the model represented in Figure 4.5. The elements comprising the model are also described below.

- **Refactoring strategy:** A specification to apply complex refactoring processes, which can be addressed to achieve a certain *goal* such as: remove a design smell, apply a high-level refactoring, etc. A *refactoring strategy* is composed of the set of parameters, which define the interface of the transformation, a precondition and a sequence of *strategy steps*.
- **Goal:** The objective to which the refactoring strategy is addressed.



Figure 4.5: *A model for refactoring strategies.*

- **Strategy query:** A function that evaluates, searches or explores the model of the system to deduce certain information from it. These functions use *system elements* as their input parameters and as their output results. *Strategy queries* are specialised into *basic queries* and *advanced queries*. *basic queries* have already been defined because they are the type of queries used in *refactorings* and *non-behaviour-preserving transformations*.
- **Advanced query:** A type of system query that is exclusively used for complex refactoring processes and therefore can be included in *refactoring strategies*:
 - *Duplicated code query:* This represents those queries addressed to identifying code duplication. Queries of this kind could be employed for example: to find out sets of code fragments which have been copied from a given one; to check whether two excerpts of code implement the same functionality, etc. The ability to obtain this information from a software system will allow us to automatically apply strategies that include refactorings such as **EXTRACT METHOD** or **PULL UP METHOD**.
 - *Numerical query:* This query represents all functions that can generate quantitative information from a software system, the most obvious example being *metrics*. These types of queries have been classified as *advanced queries* to represent the fact that they are only meant to be used in *refactoring strategies*, not in refactorings or non-behaviour-preserving transformations.
 - *User query:* Any type of information that can not be extracted automatically, but can be obtained from the developer, is still suitable to be used within a *refactoring strategy*. A user query is a special kind of query that an automated tool will launch during the instantiation of a *refactoring strategy* and has to be answered by the developer. It can include a request to make a decision about the execution path to follow when facing several options, an inquiry about a property of a *system element*, etc.
 - *Entity-kind query:* As found in some correction specifications, in some situations, it can be helpful to know the kind of *system elements* involved. For example, the most appropriate strategy to correct a design smell may depend on whether a class is a GUI class or not (see correction specification 2).
- **Strategy condition:** A more general type of condition than *boolean conditions*. We have specialised it into basic *boolean conditions* or into *quality-based conditions* to highlight the different nature of each condition type.
- **Quality-based condition:** These conditions have been found in some correction specifications. They represent conditions formulated over *software quality factors* and *software metrics*. The former are evaluated to determine whether a certain quality factor is favoured or disfavoured, while the latter will be evaluated to how a certain metric changes. These conditions could be used to guide the computation of refactoring plans, or to inform the user about the outcome of a plan, regarding quality, so the user can make additional quality-guided decisions.
- **Strategy step:** A refactoring strategy is organised into smaller steps. These *strategy steps* are similar to those that can be used in *refactorings* and *non-behaviour-preserving transformations*, but these include the possibility of using non-deterministic

algorithms that allow us to define heuristic-based refactoring processes. Each step is composed by a sequence of *strategy control* constructs. Two types of steps can be used depending on whether their component control constructs are arranged into “ordered” or “unordered” sequences. Ordered sequences of control constructs define an *ordered step*, while unordered sequences will define an *unordered step* within a *refactoring strategy*. Strategy steps should not be confused with those steps identified in current correction strategies (see Section 4.1.2). Strategy steps are atomic operations. Therefore, in our proposal, in order to organise strategies in different sub-parts, decomposition and invocation of strategies (*try* or *apply*) should rather be used.

- **Strategy control:** This defines the type of control structures that can be used in *refactoring strategies*. They can be either *deterministic* or *non-deterministic* control constructs.
- **Deterministic control:** This family of control constructs define the same ones that can be used in *refactorings* and *non-behaviour-preserving transformations*. Nevertheless, when used within *refactoring strategies*, these control constructs can make use of *advanced queries*, *quality-based conditions*, *strategy steps* and invocations for any type of *transformation*.
- **Non-deterministic control:** A family of control constructs that can be included within *refactoring strategies*. They allow us to define non-deterministic algorithms and therefore heuristic-based strategies. We define the following non-deterministic control structures:
 - **Try:** Non-deterministic invocation of a *transformation* of any kind. It represents an optional step in the strategy. Its instantiation should be tried when computing the refactoring plan. A successful instantiation would lead the invoked transformation to be part of the plan, if not, it will not be included, but still the enclosing strategy can be instantiated into a successful plan.
 - **ND Alternatives:** Non-deterministic selection of different execution paths. This structure represents a set of alternative steps. Each step is optionally guarded by a condition. For the strategy to succeed, at least one of the alternatives has to be successfully applied. In this construct, the order in which the different alternatives are tried is not specified. This allows us to write a refactoring strategy even when the ordering or the conditions are unknown or cannot be described.
 - **ND Loop:** Non-deterministic loops. When a transformation has to be iteratively applied over a set of *system elements*, the success of the process may depend on the ordering of these iterations. An *nd loop* allows us to leave this ordering unspecified, so the tool can compute the appropriate ordering that succeeds. This construct represents any type of loop that may be used, such as “while”, “foreach” loops, etc.

As a summary, it is worth discussing the differences between refactoring strategies and the current design smell correction specifications that we analysed in Section 4.1. These considerations we have taken, when defining refactoring strategies, address the problems identified in Section 4.2: applicability of refactorings and non-formal description of correction strategies.

The proposal we have introduced establishes a clear separation between the specification of complex refactoring processes, defined as refactoring strategies, and the exact refactoring sequence to apply to a system, defined as refactoring plans. This allows us to specify complex

refactoring processes, such as the correction of design smells, in a heuristic, but structured, way. At the same time, the application of the necessary refactorings is deferred to the moment when a refactoring plan is instantiated from a refactoring strategy. The identification and separation of these two concepts does not appear in the analysed current design smell correction specifications. The definition of a model for refactoring strategies allows more formal and structured specifications to be written that can be more easily reused and automated. The information included in refactoring strategies allows us to compute, in instantiation-time, the precise transformation sequences that apply the intended strategy.

We have also defined the different types of transformations involved in a complex refactoring process and we have made a clear distinction between them. All complex refactorings, including the correction of smells and high-level refactorings, are identified as refactoring strategies. This allows us to define heuristic strategies for all of them in a homogeneous way. On the contrary, we have identified non-behaviour preserving transformations and low-level refactorings as more simple transformations, which are not meant to be described in a heuristic way. We have clearly separated low-level refactorings from high-level refactorings. This allows us to state that the former cannot be composed of other refactorings. The latter, which can be composed of other refactorings, have to be treated as refactoring strategies, defined with heuristic specifications, and therefore should go through a strategy-instantiation process, prior to their execution. As a consequence of all this, it is easier to specify the simpler transformations and to refine and to reuse them as lower-level blocks to build up the more complex refactoring strategies.

To allow for heuristic specifications of refactoring strategies, we have detected the need to include non-deterministic control constructs, which we have identified, along with regular control constructs. These non-deterministic constructs are not present in the current design smell correction strategies that we have analysed. All types of transformations – refactoring strategies, refactorings and non-behaviour transformations – can include regular control constructs. Refactoring strategies, due to their heuristic nature, may also include non-deterministic control constructs. In addition to this, we have also distinguished between the basic and the advanced queries – duplicated code, numerical, user, and entity-kind queries – needed in a complex refactoring process. We have considered that the basic ones may be used in all types of transformations, but the advanced ones are only needed in refactoring strategies.

4.4 A small language to specify refactoring strategies

As an example of how our proposal can be employed, a small domain specific language (DSL) is proposed in this section. This will help us demonstrating how developers could be able to write more formal specifications for design smell correction and complex refactorings. This language also allows us to separate the refactoring strategies concept from the underlying technique that would be used to compute them. As a result, although a particular technique is proposed in this dissertation, different techniques may be used to support refactoring strategies. Moreover, with this kind of language, refactoring strategies are offered to the developers in a convenient and familiar way. They can write, improve and reuse the specifications of strategies using a simple language. They do not have to learn about the complexity and the internal details of the tool that computes plans from strategies.

Among the different types of transformations modelled in our approach –refactoring strategies, simple refactorings and non-behaviour-preserving transformations– this language is meant to be primarily used just for specifying strategies. Simple refactorings and non-behaviour-

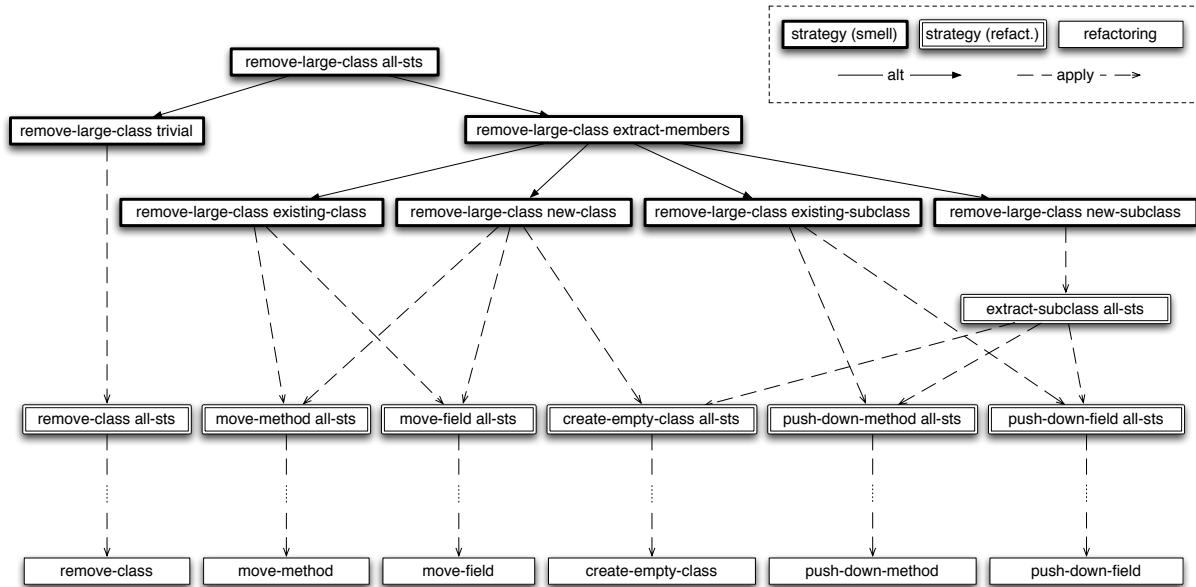


Figure 4.6: Overview of the relations between the strategies, refactorings and transformations compiled in listings 4.1 and 4.3.

preserving transformations can be invoked from strategies. Nevertheless, the details for their specification or implementation have been left out of the language. In order to further separate the language from the implementation of the approach, system elements are not meant to be accessed directly from the specifications. The information about the current state of the system is rather obtained through the invocation of queries. The definition or implementation of queries is not included in the language either, so to also enforce the isolation between refactoring strategies specifications and their underlying implementation.

In order to demonstrate how we consider this language should be, we use the running example that have already been analysed in this chapter: the strategies for removing a *Large Class* smell. A sketch of these strategies is presented in Listings 4.1 and 4.3. An overview of the relationships between the strategies and refactorings involved is portrayed in Figure 4.6. Finally, the grammar that defines the DSL is shown in Listing 4.4.

The main strategies for removing a *Large Class* are presented in Listing 4.1. The first strategy – `remove-large-class all-sts` – will serve as the main entry point to invoke all the available strategies. In order to do that, its body consists of just an `alt` construct, where each alternative branch contains an invocation of a different strategy. In this particular case, two strategies have been written. As more heuristics are compiled by developers, new strategies can be written and more branches would be added.

In this example, the first strategy – `remove-large-class trivial` – represents the trivial solution that attempts to simply remove the smelly class. It is guarded by the condition `is-unreferenced-class (package, class)`, so the `remove-class` refactoring may only appear in the plan if there is no reference to the class in the current system. The condition of this branch could have also been moved to the precondition of the trivial strategy. As shown in

```

strategy remove-large-class all-sts (package, class)
  precondition

  body
    alt
      branch is-unreferenced-class (package, class)
        apply remove-large-class trivial (package, class)
      branch
        apply remove-large-class extract-members (package, class)
      end
    end
end

strategy remove-large-class trivial (package, class)
  precondition

  body
    apply remove-class all-sts (package, class)
  end

strategy remove-large-class extract-members (src-package, src-class)
  precondition

  body
    user-query("Which members to extract?", members)
    user-query("Which target class?", tgt-package, tgt-class)
    alt
      branch
        apply remove-large-class existing-class (src-package, src-class, members
          , tgt-package, tgt-class)
      branch
        apply remove-large-class new-class (src-package, src-class, members, tgt
          -package, tgt-class)
      branch
        apply remove-large-class existing-subclass (src-package, src-class,
          members, tgt-package, tgt-class)
      branch
        apply remove-large-class new-subclass (src-package, src-class, members,
          tgt-package, tgt-class)
      end
    end
end

```

Listing 4.1: *Draft of the top strategies that may be defined to remove a large class.*

```

strategy remove-large-class all-sts (package, class)
  precondition

  body
    alt
      branch
        apply remove-large-class trivial (package, class)
      branch
        apply remove-large-class extract-members (package, class)
    end
  end

```

Listing 4.2: *An alternative version of the main strategy for removing a Large Class.*

the alternative strategy of Listing 4.2, the condition can even be removed completely and the `remove-class` refactoring can be invoked directly. The semantic of the `alt` construct implies that we let the tool to find out the suitable alternative. The query `is-unreferenced-class` is surely included within the precondition of the `remove-class` refactoring, so the tool should find by itself that the trivial strategy cannot be applied.

The second strategy – `remove-large-class extract-members` – comprise all the strategies that undertake removing the smell by extracting some members from the Large Class to another class. We haven't found an appropriate condition for this branch, therefore no condition has been given to it. As a consequence, the tool should tackle the computation of this branch using the knowledge compiled in its corresponding substrategies, or in the refactorings used by them.

The different substrategies that try to extract members away from the smelly class are invoked from the `remove-large-class extract-members` strategy. Four alternative substrategies are going to be checked depending on where the extracted members shall be moved to: `remove-large-class existing-class`, `remove-large-class new-class`, `remove-large-class existing-subclass` and `remove-large-class new-subclass`. All these strategies require some additional information: which are the members to extract and which class should they be moved to. We are only demonstrating how refactoring strategies can be used by a developer, thus we have simply included two user queries for gathering this information before invoking the substrategies. A user query takes a message, that will be shown to the developer, as an input argument and the variables, that will hold the information provided back, as output arguments. Procedures such as the RCA approach for computing cohesive sets [Moh08], which has been previously mentioned in Section 4.1 “Other related works”, can be used in the form of queries in order to support a more automated way to select the members to be moved and therefore a more comprehensive strategy to remove the *Large Class* smell.

It should be noted how the invocation of strategies can be distinguished from simple refactorings or transformations. Strategies are defined and invoked by a name and a goal. For example, `remove-class all-sts` refers to a strategy whose goal is to apply a **REMOVE CLASS** refactoring, and whose name – `all-sts` – suggests that it is the top strategy for that goal. On the other hand, invoking just `remove-class`, would be interpreted as the **REMOVE CLASS** refactoring itself.

A strategy is identified by its goal, name and number of arguments, while refactoring and non-behaviour-preserving transformations are identified by a name and their number of arguments. It is allowed to define different strategies using the same name, number of arguments and goal, as to use different refactorings and transformations with the same name and number of arguments. The invocation of this multiple-defined transformations, should have an effect similar to that of the `alt` construct: when computing a refactoring plan, the tool should non-deterministically select one of them at a time, and try to instantiate it to be part of the resulting plan, until one of them succeeds or until all of them fail.

There is another aspect of invocations which is worth mentioning. In our proposal for refactoring strategies (see Section 4.3) we consider two different types of invocations: a deterministic and a non-deterministic one: ***apply*** and ***try***. Obviously, the language allows both of them to be used. Although the listings presented as illustrative examples only show the `apply` invocation, the `try` invocation can be used too. The `apply` invocation forces the tool to compute a successful instantiation for that invocation, which will be included in the resulting plan, or else to fail. The `try` invocation represents an optional part of the plan and its instantiation is non-deterministic. The tool should try to perform it, but regardless of a successful instantiation being obtained or not, the computation of the refactoring plan should continue.

The Listing 4.1 can also be used to discuss other aspects of the language. The evaluation of queries is meant to be similar to those of PROLOG. Variables already associated to a value take the role of input parameters, while those variables without an assigned value will work as output parameters. Queries are always interpreted as boolean expressions, so the result of an evaluation of a query is a boolean value. When the resulting value is `true`, all the parameter variables will have values associated to them. If a suitable result for the unassigned variables cannot be found the query cannot be evaluated to `true`. Thus, the step could not be performed and the tool should backtrack to the last decision point in order to choose another path.

In Listing 4.1 we can also identify the basic tokens of the language, which are: symbols, literals and reserved words. Symbols are used for variables and for naming and invoking strategies, refactorings, transformations and queries. Literals can be either numeric literals, belonging to reals, or string literals enclosed in double quotes. The reserved words of the language, highlighted in bold font in Listings 4.1 and 4.3, are: `strategy`, `precondition`, `body`, `apply`, `try`, `alt`, `branch`, `if`, `elseif`, `else`, `while`, `foreach`, `in`, `satisfying`, `end`, `unordered`, `not`, `imply`, `or`, `and` and `forall`.

The strategies proposed for extracting members from the *Large Class* are shown in Listing 4.3. The purpose of these strategies is to specify which refactorings must be applied for each case and how. These strategies invoke other strategies, whose objectives are to apply different refactorings successfully. The relationships between all these transformations and the type and purpose of them are graphically depicted in Figure 4.6. It is worth noticing that **EXTRACT SUBCLASS** is a complex refactoring and therefore, there is not a simple refactoring defining or implementing it. Instead, it should be specified as a refactoring strategy and thus it has been represented as such in Figure 4.6. It will probably be composed from other simpler refactorings, such as **CREATE EMPTY CLASS**, **PUSH DOWN METHOD** and **PUSH DOWN FIELD**, so this is also depicted in Figure 4.6.

The strategies of Listing 4.3 can illustrate how the control structures included in our proposal can be used within the specification of a strategy. In the first strategy – `remove-large-class existing-class` – a `foreach` loop is used to iterate over all the members to be extracted from the *Large Class*. The name of these members, a collection of literals, should have been passed

```

strategy remove-large-class existing-class (src-package, src-class, members,
tgt-package, tgt-class)
  precondition
    exists-class(tgt-package, tgt-class)
  body
    foreach member in members
      unordered loop
        if (is-field(package, class, member))
          apply move-field all-sts (package, class, member)
        elseif (is-method(package, class, member))
          apply move-method all-sts (package, class, member)
        end
      end
    end
end

strategy remove-large-class new-class(src-package, src-class, members, tgt-
package, tgt-class)
  precondition

  body
    apply extract-class all-sts (src-package, src-class, members, tgt-package,
tgt-class)
end

strategy remove-large-class existing-subclass (src-package, src-class, members
, tgt-package, tgt-class)
  precondition
    inherits(tgt-package, tgt-class, src-package, src-class)
  body
    foreach member in members
      unordered loop
        if is-field(package, class, member)
          apply push-down-field all-sts (src-package, src-class, member, tgt-
package, tgt-class)
        elseif is-method(package, class, member)
          apply push-down-method all-sts (src-package, src-class, member, tgt-
package, tgt-class)
        end
      end
    end
end

strategy remove-large-class new-subclass(src-package, src-class, members, tgt-
package, tgt-class)
  precondition
    not (inherits(tgt-package, tgt-class, src-package, src-class))
  body
    apply extract-subclass all-sts (src-package, src-class, members, tgt-
package, tgt-class)
end

```

Listing 4.3: *Draft of some strategies that may be defined to remove a Large Class.*

to the strategy as the parameter `members`. Each iteration, the variable `member` is assigned a different value from the variable `members` and the body of the loop construct is computed with it. The loop has been labelled as `unordered`. This means that, for all the possible orderings of the iteration over the elements of `members`, the tool should attempt to instantiate the body of the loop into a part of a refactoring plan until one of the orderings succeeds or until all of them fail. For regular loops, not labelled as `unordered`, only the first ordering will be checked.

We have also considered it necessary to include in the language other types of loops. They do not appear in the illustrative listings so they are described here. A more complex kind of “foreach” loops `-foreach (vars) satisfying (cond)-` will be used to iterate over tuples of values that make `cond` evaluate to true when each tuple of values is assigned to the variables appearing in the variable list `vars`. The loop header is defined with a list of variables `-vars-` and a condition `cond`. The tool should find all the combinations of values, for the variables in `vars`, that would satisfy the condition `cond`. An iteration should be run for each distinct tuple of values. Each iteration, the variables in the list `vars` would be assigned a different tuple of values. This more complex “foreach” is similar to the regular one. For the regular “foreach”, a collection has to be computed first, and the loop iterates over a single variable which takes the value of each element in the collection in each iteration. This more complex “foreach” works in a similar way but in this case, the collection is built dynamically and each element can be a tuple rather than a single variable. The collection is built by gathering all tuples satisfying the given condition. Finally, a more simple “while” loop `-while (cond)-` can be used to repeat the body of the loop while the condition `cond` evaluates to true.

The `if ... elseif ... else` construct is a deterministic multiple-alternatives structure. The first branch must be introduced with `if`, the optional middle branches with `elseif`, and an optional default branch, not shown in the example, with `else`. A branch should only be computed if the condition of the previous branch fails. If the condition of a branch can be evaluated to true, its body will be computed to instantiate a part of the plan, and the other branches will be discarded regardless of that instantiation being successful or not. As opposite to the `alt` construct, the branches should be evaluated in the specified order and their associated conditions are not optional but mandatory. The default branch, introduced with the `else` reserved keyword, is not guarded by a condition. In short, this structure works as a regular “if-then-else” conditional structure. In the case of the example shown in Listing 4.3, at this point of the strategy we have complete confidence that a member should be either a field or a method so a non-deterministic alternative structure is not needed. We can save computation time, and avoid trying multiple or non-deterministic options if we have confident and complete knowledge about a specific situation and therefore we are able to include it in the strategy.

The full specification of the DSL proposed for writing refactoring strategies is finally presented in Listing 4.4 in Extended Backus-Naur Formalism (EBNF).

4.5 Characteristics of the problem

This section revisits the problem of planning complex refactoring sequences and analyses its main characteristics. Once we have proposed a model and a language for writing refactoring strategies, this brief analysis prepare the way for developing an approach that could automate the computation of refactoring plans from refactoring strategies. Our approach to tackle this problem, relies on these two requirements that have to be addressed:

<i>strategy</i>	::=	strategy goal name <i>'('args?')'</i> prec? body end
<i>goal</i>	::=	symbol
<i>name</i>	::=	symbol
<i>args</i>	::=	arg <i>'('</i> arg <i>*)</i>
<i>arg</i>	::=	atom <i>'('args'</i>
<i>prec</i>	::=	precondition cond
<i>body</i>	::=	body step*
<i>step</i>	::=	query invocation compound-step
<i>query</i>	::=	name <i>'('args'</i>
<i>invocation</i>	::=	(try apply) goal? name <i>'('args?')'</i>
<i>compound-step</i>	::=	loop alt unordered-block
<i>loop</i>	::=	loop-header (unordered)? loop vars* step* end
<i>loop-header</i>	::=	while cond foreach var in var foreach vars satisfying cond
<i>var</i>	::=	symbol
<i>vars</i>	::=	var <i>'('</i> var <i>*)</i>
<i>alt</i>	::=	alt (branch cond? step*)+ end if cond then step* ((elseif cond then step*)* else step*)? end
<i>cond</i>	::=	<i>'(cond'</i> not cond cond or cond cond and cond cond cond query
<i>unordered-block</i>	::=	unordered step* end
<i>atom</i>	::=	symbol literal
<i>literal</i>	::=	"s"; s ∈ string n; n ∈ double
<i>symbol</i>	::=	letter (digit letter <i>'_'</i> <i>'-'</i>)*
<i>letter</i>	::=	[a–z] [A–Z]
<i>digit</i>	::=	[0–9]

Listing 4.4: Grammar for the refactoring strategy DSL.

1. A proposal to write automation-suitable specifications of complex refactoring processes.
2. An automated support to plan ahead the computation of complex refactoring sequences.

The first requirement has been addressed by defining refactoring strategies. This concept unifies the existing ways of writing design smell correction specifications and allows us to write heuristic-based specifications that can be automated. In order to automate the computation of complex refactoring sequences, we must now analyse the problem of instantiating refactoring strategies into refactoring plans.

Computation. To apply a refactoring strategy, the automated tool should take into account the dependencies and conflicts between the refactorings, refactoring strategies and non-behaviour preserving transformations that compose that strategy. These computations, to find out which transformations enable or disable others, can be performed by examining and operating the preconditions and postconditions of the transformations available. Since specifications of transformations do not usually include postconditions, but only preconditions and the mechanics of the transformations themselves, it is difficult to put this approach into practice. This would require writing postconditions or deducing them automatically. Another approach to perform this computation would use preconditions and transformation descriptions and will have to be performed by applying the transformation or by simulating it to find out its effects. Therefore, a model of the system is needed, as well as an automated support that has to be able to compute preconditions and to simulate transformations over this model.

Software model. As already proposed in the definition of refactoring strategies, a model of the system can comprise, in a general way, system elements and relationships. Moreover, we have to determine which is the most adequate level of detail needed to represent a software system for the purposes of this dissertation. The execution of refactorings have to be underpinned by changes to this model. Most refactorings, both their precondition and their mechanics definition, work at below-method level. Therefore, to support as many refactorings as possible, the system model has to present a level of detail similar to that of the source code. The system has to be represented at statement and expression level. An AST-based model, for example, will do.

Deterministic and non-deterministic control constructs. If the approach to compute refactoring plans has to find out how to schedule refactorings, transformations and refactoring strategies, by applying it over a model, it is crucial that this approach supports the control constructs that have been identified in the definition of refactoring strategies. Transformations are algorithmic procedures that include regular control constructs, such as alternatives, loops and invocations of other transformations. We have also identified the need to support non-deterministic control. Consequently, the approach should allow for simulating the execution of transformations that include both types of control constructs.

Incomplete specifications. Also related to non-determinism, the approach should support the computation of refactoring plans when developers have managed to specify all the necessary knowledge on how to perform a refactoring strategy. If this knowledge is available, the tool should take advantage of it. Nevertheless, it should be noticed that this knowledge is heuristic-based, and that the specifications of refactoring strategies are designed to be incrementally improved over time. They become more detailed and formalised by compilation of additional, more detailed

and more precise empirical knowledge. As already reviewed, specifications of complex refactorings and correction strategies present different degrees of detail and formality. Therefore, the tool should still be able to compute the possible refactoring plans when the available knowledge, on how to perform a complex refactoring process, is incomplete or not detailed enough. Indeed, this is the most common case.

Elements of refactoring strategies. As a summary, the approach to instantiate refactoring plans from refactoring strategies should support all the elements defined in refactoring strategies, such as: invocation of refactorings, non-behaviour-preserving transformations and refactoring strategies, complex system queries, complex preconditions, regular control constructs, non-deterministic control, and instantiation of refactoring plans in the case of incomplete knowledge about the complex refactoring process. This approach is presented in the next chapter.

Chapter 5

Refactoring Planning

This chapter focuses on the technique we propose for supporting refactoring strategies and refactoring planning. We claim that the problem of refactoring planning for the correction of design smells can be modelled and addressed as an automated planning problem. More precisely, we tackle the problem using **Hierarchical Task Network (HTN) planning**. Within this approach, most of the issues in the refactoring planning domain are typical issues already addressed within the automated-planning community.

In order to take into consideration issues not covered by automated planning techniques, those will be abstracted as domain knowledge which will have to be added to the refactoring planner. As a consequence, all the considerations in the refactoring planning problem are expected to be incrementally implemented and refined. This means that, after prototyping a basic refactoring planner, most of the work will be aimed at developing and implementing heuristics to progressively improve the planner.

This chapter compiles a brief introduction on automated planning, an argumentation on the particular automated planning approach that has been selected and a formalization of the refactoring planning problem.

5.1 Lessons learned from previous works

We explored other techniques, such as graph transformation, before developing the automated planning approach presented in this Thesis. A summary of this previous approach is included here because of the lessons learned from it.

We first searched for an adequate formalism for supporting the refactoring planning problem. The Intrinsic characteristics of refactorings include the modification of the system structure and the execution of behaviour preserving transformations. Hence, we hypothesized that the formalism should be suitable for the description and manipulation of structural information. It should also allow precondition checking to be represented for the transformations modelled with it. Graph transformation [Roz97, EEKR99, EKMR99] was the general formalism selected for the first stages of this research. Previous works from other authors demonstrated the validity of the graph transformation approach for refactoring formalization [EJ05, MVDJ05]. In these works, programs and refactorings are represented with graphs in a language independent way, using a kind of abstract syntax trees with the appropriate expressiveness level required for the problem. This type of representation, called *Program Graphs*, was used as the basis for developing a specific graph representation for this research.

Start Graph:	Original System
Stop Graph:	Desirable Design
Production Rules:	Refactorings
Language:	Set of software systems equivalent to the original system
Production Path:	Refactoring plan

Table 5.1: *Interpretation of the refactoring planning problem as a graph grammar problem.*

The problem of refactoring plan generation was introduced in [Pér05] and presented with more detail in [Pér06]. We assumed the existence of a reference structure or design, to which we want to migrate the original system. We applied searching algorithms to find refactoring sequences between the original system and a fictional redesigned target, matching that reference structure.

Formal language problem

A preliminary approach to the refactoring plan generation problem was based on graph grammar parsing and was presented in [Pér05]. Our first intention was to model the problem as a formal language problem. The problem of refactoring planning was interpreted as depicted in Table 5.1. The redesign problem is seen as a graph rewriting system whose language is the set of those software systems behaviourally equivalent to the original one, which are reachable from this original system by using the available refactorings as graph production rules. The problem of checking whether the target desirable system could be derived or not from the source system using the available set of defined refactorings, was interpreted as the membership problem of formal language theory. The problem of computing the refactoring plan was addressed as obtaining a rule sequence that would produce the stop graph from the start graph.

This approach was addressed by taking advantage of the graph parsing capabilities offered by some graph grammar systems. More specifically, AGG [ERT99, Tae03, agg] was the graph transformation tool used in our experiments. A graph grammar parsing tool can compute whether a certain graph belongs to a graph language and, consequently, it may also be able to compute the sequence of rules which can be used to derive that graph. The graph language is defined by the start graph and the production rules. Graph language parsing is manageable for restricted graph grammars [RS97, BTS00], such as layered graph grammars. AGG supports these grammars, therefore, we tried to formulate refactorings in such a way that they could fit as a layered-graph-grammar [Pér05]. Unfortunately, these initial explorations were of no practical use. Graph grammar parsing deals well with visual language parsing [BMST99]. In these problems, the derivation paths are well defined. Unfortunately, for the refactoring discover problem, the number of different instances of different refactorings which can be applied over a software system for each derivation step is huge and almost unpredictable. We could not find a way to formulate refactorings so they could constitute a layered graph grammar with a well known, precise and non-ambiguous derivation tree. Therefore, we abandoned this approach.

The experience of trying to write refactorings as graph transformation rules helped us understand the refactoring planning problem better. The graph transformation rules needed to specify refactorings are rather complex, specially if we want to represent and manipulate the software system at the level of detail of the methods' bodies. Graph grammars do not suffice to

Start State:	Original System
Goal State:	Desirable Design
State transitions:	Refactorings
State Space:	Set of software systems equivalent to the original system
Search Problem Solution:	Refactoring plan

Table 5.2: *Interpretation of the refactoring planning problem as a state-space search problem.*

define refactorings at this level of detail. Some kind of programmed control would be needed. We also noticed that the language independent *Program Graphs* representation formalism had to be extended with language specific elements for it to be usable for representing real refactorings. We also found out that a huge number of transformations can be applied at each derivation step. This makes the problem a search problem with an infinite state space. Therefore, it should be addressed with some degree of guidance. On the basis of these conclusions, we explored another direction, based on graph transformation as well.

State-space search guided with postconditions

We then tried to tackle the refactoring planning problem as a state-space search problem [PC07a]. In order to allow running experiments for real systems, we developed an extension to *Program Graphs* called *Java Program Graphs*. It included JAVA concepts such as visibility, interfaces and packages to support the representation of real software systems and refactorings. Within this approach, we interpreted the refactoring problem as summarised in Table 5.2. We addressed obtaining a refactoring plan as a computation of the solution for a state-space search problem, where the start state would be the original system, the goal state would represent the desirable redesigned system and the state transitions would match refactoring specifications. We identified the problem of whether a refactoring plan exists as a reachability problem. The refactoring plan would be extracted from the search problem solution as the path from the start state to the goal state.

We explored basic search algorithms to look for refactoring sequences. In order to allow some kind of guided search, we based our solution on the use of refactoring preconditions and postconditions. Therefore, the approach required refactoring definitions, including preconditions and postconditions, in their specification. The main idea of our algorithm was to iteratively modify the start state by applying refactorings written as graph transformation rules. At each iteration, the set of selectable refactorings was composed of those refactorings whose preconditions held in the current state and whose postconditions held in the goal state. When no more refactorings were selectable, the algorithm backtracked to the last transformation applied. The algorithm would succeed when the current state graph was isomorphic to the goal state graph. The refactoring sequence then matched the path found to the goal state. The algorithm would fail when no more refactorings could be executed, and the current and goal states were not isomorphic.

We developed an initial prototype as an Eclipse plugin [MA06, PC07b], in which we used the AGG graph transformation tool as the back-end. We selected AGG mainly because it supports graph parsing, and is easy to use, thus allowing rapid prototyping. Graph parsing can be used to perform depth-first search with backtracking, and our algorithm could be partially implemented in that way. The AGG support for layered rules was also useful for implementing

some programmed control. Our tool implemented breadth-first search, depth-first search and breadth-first search with iterative deepening.

We were able to run some toy-experiments with this tool as a proof of concept. Nevertheless, we were not able to get successful results with real software systems. The approach did not scale to problems with systems larger than 20 classes. We noticed that we could not find a way to take advantage of the graph transformation formalism. We were able to employ graph transformation tools merely as inefficient transformation execution engines that do not scale to our problem. We also found the dependency on a reference design to be a major problem. The tool can only determine that the goal state has been reached by checking the isomorphism between the current graph and the stop graph. This implies the stop graph has to be a full representation of the desired redesigned system and, therefore, there is no point in searching for a refactoring plan when the redesigned system is already available.

Lessons learned

Along with our previous works, we elaborated a case study to determine the feasibility of graph transformation for representing and implementing real refactoring specifications and thus, to decide whether it can be the basis for the refactoring planning problem [PCHM10]. The case study was addressed by us and other authors with different approaches based on graph and model transformation tools [AL08, Gei08, MDBJ08, PRT08, Wei08, MJ10, JBK10, MSF⁺10, GZ10]. Although graph transformation is theoretically suitable as an underlying formalism for the specification and implementation of refactorings, we have concluded that it currently lacks the necessary tool support for refactoring planning. More precisely, we could not find a tool with all the required features: enough expressiveness to allow for any refactoring specification, support for computing or representing, the dependencies and relations between refactorings, and more importantly, an efficient and scalable transformation approach and environment to execute refactoring specifications while performing a state-space search process.

One of the biggest drawbacks of our graph-transformation-based approaches was the assumption of the existence of a reference design or a desirable restructured version of the system. We finally concluded that the availability of this target system cannot be relied upon. On the contrary, the empirical knowledge on how to redesign a software system by performing complex refactoring processes is usually gathered empirically and given in terms of refactoring “recipes”. The most detailed refactoring “recipes” are Trifu’s reorganization strategies. Unfortunately, they can only be found in his PhD Thesis dissertation [Tri08], and are available for a small amount of very specific problems. More general refactoring “recipes” can be found more extensively in the literature. We concluded that a more practical approach for refactoring planning should rather be based on this kind of knowledge than on our former reliance on the existence of a restructured version of the software system.

According to a study by Mens *et al.* [MKR06], graph transformation and logic programming offer similar capabilities for representing software transformations and for performing transformation dependency analysis. Nevertheless, the computation mechanism of PROLOG, unification, is sufficient for this problem and is more lightweight than graph transformation tools. Logic programming approaches seem to offer better expressiveness than graph transformation techniques and a huge performance improvement –over 3 orders of magnitude in the referenced study. We also concluded that the characteristics of the refactoring planning problem make it appropriate for being addressed as a state-space search problem. Due to the huge size of the state-space, our efforts had to be directed to applying an efficient search-based technique and to transferring the

World's state:	list of terms
Operators:	definition: name + arguments precondition effect list (add): terms to add to the state effect list (delete): terms to remove from the state
Problem:	initial state goal: list of terms
General planning approach:	chain op. by matching effects and preconditions

Table 5.3: *Classical Planning Operators (STRIPS)*

available knowledge about how to remove bad smells to proper heuristics to guide the search. Further exploration of these subjects –logics and state-space search– led us to an approach based on automated planning [Pér08, PC09], which is the one described in this Thesis.

5.2 A brief introduction to automated planning

Automated planning [GNT04] is an artificial intelligence technique aimed at computing the sequences of actions that will achieve a certain goal when they are performed.

In automated planning, the problem domain is known as “the world” and each current situation of the problem is known as “the state of the world”. This state is made up of “facts”, which include problem entities and instances of the relations between them. For a classical automated planner (see Figure 5.3), these facts are usually specified with first order logic predicates which describe the affirmations that are true or false at a particular state of the world. The actions that change the world are modelled with “operators”. During the planning process, the set of logic terms which build up the state of the world is changed through the application of these operators. The plan we want to obtain is a specific sequence of operators that can be applied in order to achieve the goal state. The planner has to reach a state in which the goal facts hold. A simple planning problem is composed of an initial, or start, state, a goal and the set of available operators.

The basics of automated planning can be illustrated by using an introductory example: how to go shopping for apples and a book. The problem can be modelled by representing the state of the world with two different predicates: `at(X)`, `have(X)`. The entities in the problem: `home`, `grocery`, `bookstore`, `apples`, `book`, can be used to substitute the variables in those predicates to describe a particular state of the world. For example, we could describe the initial state of the world as: `at(home)`. The goal can be represented as: `at(home) AND have(apples) AND have(book)` and operators would be represented with terms such as: `buy(X)`; `moveTo(X)` which would be eventually instantiated to predicates such as `buy(apples)`; `moveTo(bookstore)`.

In order to compute the valid plans, an automated planner has to be fed with the knowledge of when the operators can be applied and how they change the world's facts. The classical way to represent these operators in a planning problem is in the form of STRIPS operators (see Table 5.3), named after their development for the STRIPS automated planner [FN71].

Operators, have to be specified in terms of an operator definition (its name and parameters), a set of conditions which build up the operator precondition and a set of actions which add or delete relationships (terms) in the world's state.

Operators:	
definition:	moveTo (X)
precondition:	at (Y) , not (at (X))
effects:	
add:	at (X)
delete:	at (Y)
<hr/>	
definition:	buy (X)
precondition:	not (have (X)) , at (Y) , sells (Y, X)
effects:	
add:	have (X)
delete:	
<hr/>	
Initial state:	at (home) , sells (grocery, apples) , sells (bookstore, book)
Goal state:	at (home) , have (apples) , have (book)

Table 5.4: *Specification of the apples and book problem.*

The precondition of an operator specifies the conditions under which it can be applied. The two separate sets of actions specify how the operator modifies the state of the world. To do so, these sets enumerate the terms the operator will add to the current state and the terms it will delete from it.

A goal is a list of terms representing a certain state of the world that we want to achieve. A planner computes a plan as a sequence of instantiated operators that change the world in a way that the desired goal holds in the final state. A planning problem is composed of three sets of terms representing the set of operators, a world's state and a set of goals to achieve. Once a planning problem has been fed to a planner, we can ask the planner for a sequence of operators that lead the world from some initial state to another goal state. This goal state, does not need to be a full description of the world, but a set of conditions/goals that we want the goal state to hold.

By matching effects and preconditions between the available operators, a planner can decide which steps have to be included in a plan and which should be their order. If an operator has been selected to be part of the plan and its precondition does not hold, it must be preceded by another operator whose effects make that precondition true.

To fully specify the apples and book problem, according to this classical approach, the complete definition of its operators is shown in Table 5.4. A planner will search the state space of this problem, looking for a state in which the goal is true. The resulting plan will be any sequence of instantiated operators that transforms the initial state to a state in which the goal holds.

The complete planning state graph for the apples and book problem is shown in Figure 5.1. Any path that reaches the goal state – at (home) AND have (apples) AND have (book) – from the initial state at (home) will produce a valid plan. For example, one of the two shortest possible paths will be: moveTo (grocery) ; buy (apples) ; moveTo (bookstore) ; buy (book) ; moveTo (home) . This path is highlighted in Figure 5.1 with bold arrows.

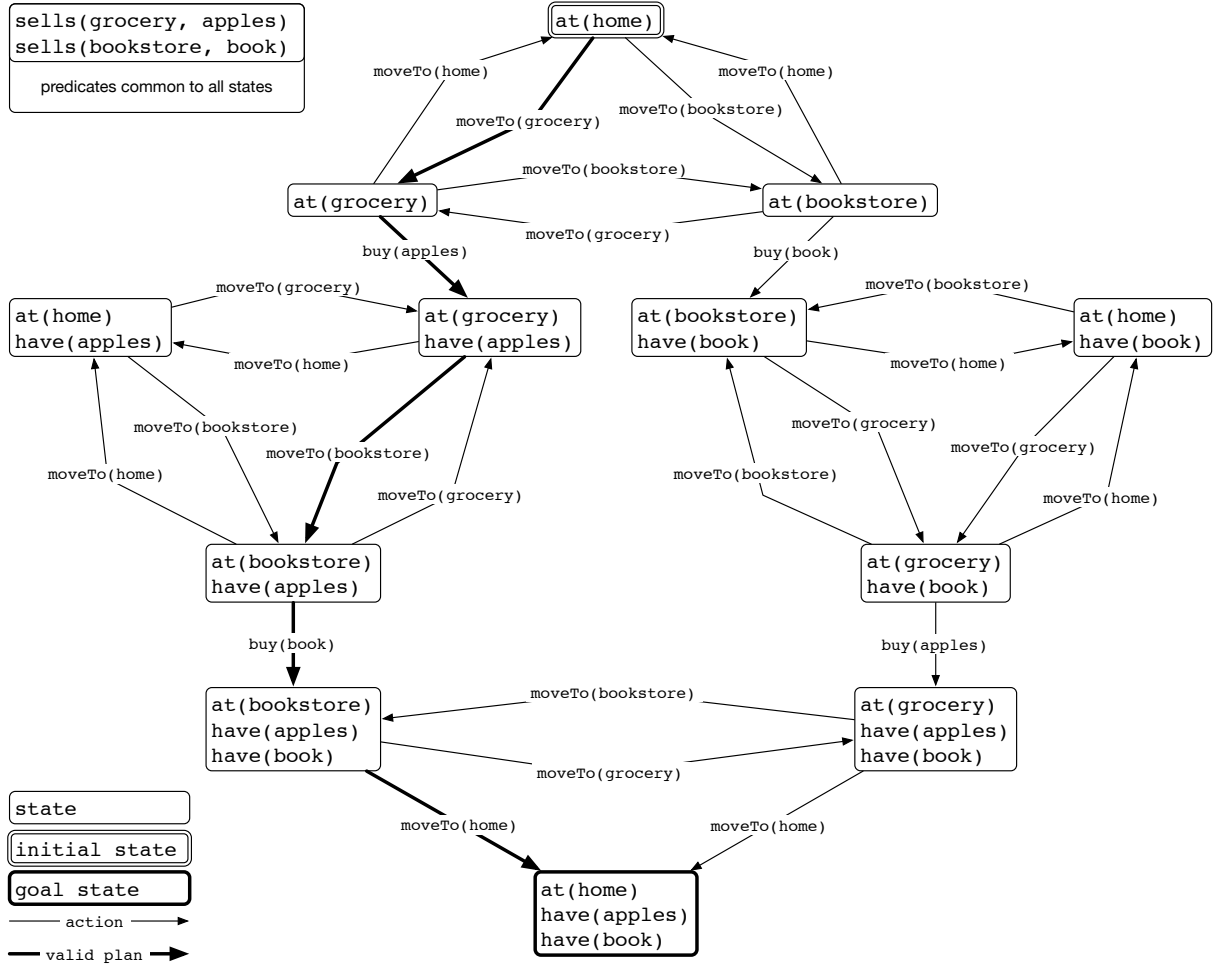


Figure 5.1: Complete planning state graph for the apples and book problem. The `sells()` predicates are common to all states and thus, are omitted from the state rectangles for the sake of simplicity.

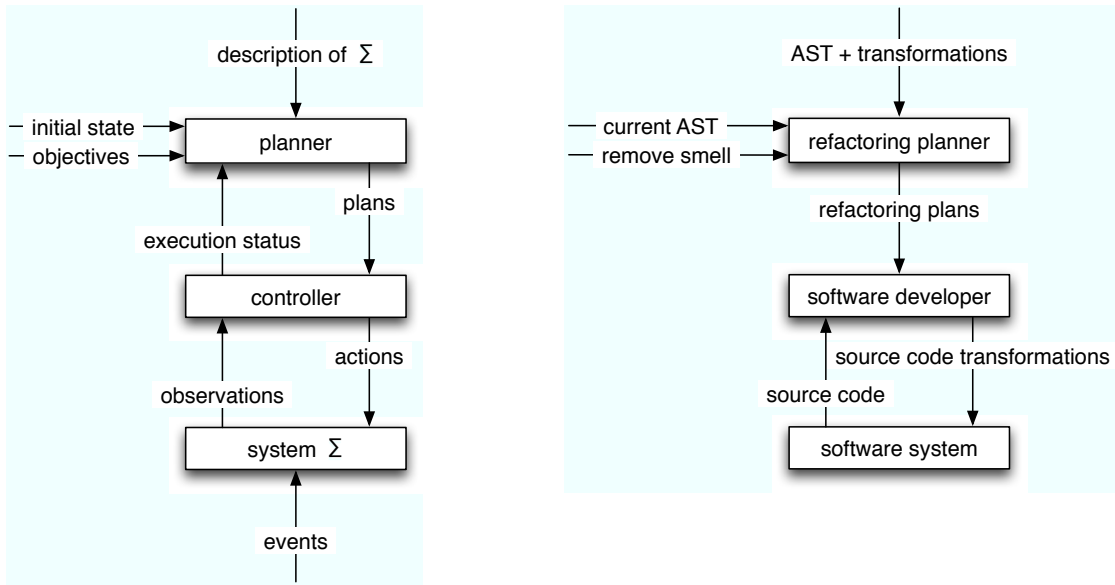
5.2.1 Formal basics of automated planning

To define the basics of automated planning, a general conceptual model can be used¹. Since planning is about how to change the state of a system through actions, a state transition system serves as a general model. A state transition system is an abstract machine whose dynamics can be represented with a set of discrete states and a set of transitions between these states.

Definition 1. A state-transition system is a 4-tuple $\Sigma = (S, A, E, \gamma)$, where:

- $S = \{s_1, s_2, \dots\}$ is a finite or recursively enumerable set of states.
- $A = \{a_1, a_2, \dots\}$ is a finite or recursively enumerable set of actions.

¹We have extracted all the formal descriptions and definitions of the basics of planning and HTN planning from [GNT04]



(a) The conceptual model for dynamic planning³. The simpler model, for a static planner, will not include the “execution status” feedback from the controller.

(b) The conceptual model for the refactoring planning problem. The system does not receive nor react to events and the planner does not get execution status from the controller either.

Figure 5.2: Conceptual models for dynamic planning and the refactoring planning problem.

- $E = \{e_1, e_2, \dots\}$ is a finite or recursively enumerable set of events, and
- $\gamma : S \times A \times E \rightarrow 2^S$ is a state-transition function.

Actions are controlled and triggered by the plan executor, while events belong to the internal dynamics of the system. The planner has no control over events, although they may have to be considered for the planning process to succeed. Both actions and events contribute to the system transitions. If we represent the system as a transition graph, the edge that represents the transition from s , to s' , where $s, s' \in S$, will be labelled as u , u being a pair (a, e) , where $a \in A$ and $e \in E$, and $s' \in \gamma(s, u)$. A neutral event ϵ and a neutral action λ are introduced so the transition $\gamma(s, a, \epsilon)$, due to actions only, can be written simply as $\gamma(s, a)$, and the transition $\gamma(s, \lambda, e)$, due to events only, can be written simply as $\gamma(s, e)$.

The basic model for a planning system is depicted in Figure 5.2(a). This model is composed of: the state-transition system Σ , that reacts to events and actions and evolves as defined by the transition function γ ; a controller, that observes the current state s of the system and sends actions to the system according to a plan; a planner, that uses a description of the system Σ , a representation of the initial state, the specification of an objective and produces a plan for the controller to achieve that objective. In the case of dynamic planning, the planner receives the execution status from the controller, otherwise, the only information the planner gets about the current situation of the system is its initial state.

The controller usually does not have the complete knowledge of the system. The observation of the controller is a function $\eta : S \rightarrow O$, that maps S into a set $O = \{o_1, o_2, \dots\}$. The function

³The conceptual model and its graphical representation have been extracted from [GNT04, Page 8].

η models the partial knowledge that the controller has of the system Σ . Given a current state s , the controller receives as input the observation $o = \eta(s)$.

Given a state transition system Σ , the purpose of planning is to decide which actions to apply to the system in order to achieve a certain objective, starting from a particular initial state. This objective can be formulated in different ways:

- **with a set of goal states:** Given a goal state or a set of goal states, the objective is to select a sequence of actions that will make the system transit from an initial state to one of the goal states.
- **with a condition over the sequences of states:** The objective is to find a sequence that holds some condition over the states transitioned, such as avoid, reach or stay in particular states.
- **with a utility function:** The sequence of states traversed produces a value for a utility function and the objective of the problem includes the optimization of this function.
- **with a set of tasks:** Define the objective as tasks that the system has to perform. These tasks can be recursively decomposed into actions and other tasks.

5.2.2 The restricted model of classical planning

With these basic definitions, the eight assumptions used in classical planning to restrict and characterise planning problems can be presented. The simplest case for a planning problem combines the eight assumptions, and is called the restricted model.

Assumption A0 (finite Σ): The system σ has a finite set of states.

Assumption A1 (fully observable Σ): The system σ is fully observable. The function η is the identity function. We have complete knowledge of the current state of the system Σ .

Assumption A2 (deterministic Σ): The system is deterministic. For every state, each action applicable to that state produces a transition to another single state.

Assumption A3 (static Σ): The system receives no events. The system has no internal dynamics and remains static until an action is applied.

Assumption A4 (restricted goals): The planner only handles restricted goals, that are expressed explicitly as a goal state s_g or a set of goal states S_g . Goals that include forbidden paths or states are excluded.

Assumption A5 (sequential plans): A solution plan to a planning problem is an ordered finite sequence of actions.

Assumption A6 (implicit time): Time is not handled. Transitions are meant to be instantaneous.

Assumption A7 (offline planning): The planner is not concerned with any change that may occur to the system during the planning process. The planner only takes into account the initial and the goal states and disregards systems dynamics if they exist.

Different planning approaches have to be used depending on which assumptions the planning problem holds. Classical planning approaches can only tackle restricted problems that meet all the eight assumptions. On the contrary, modern planning approaches deal with problem domains for which some of the eight assumptions are not fulfilled. For example, problems for which non-deterministic actions are allowed –assumption A2 is not met– are addressed by *Planning with uncertainty*. Problems for which time is relevant –assumption A6 is not held– are addressed by *Planning with time* approaches. An overview on different planning approaches has been compiled in Section 5.2.4. A system that meets those restrictions, can be defined more simply as $\Sigma = (S, A, \gamma)$ instead of $\Sigma = (S, A, E, \gamma)$, since there are no events changing the system’s state. For such a system, the planning problem can be simply defined.

Definition 2. Given the restricted model of classical planning, the problem of planning can be expressed as: given a system $\Sigma = (S, A, \gamma)$, an initial state s_0 and a set of goal states S_g , find a sequence of actions $\langle a_1, a_2, \dots, a_k \rangle$, corresponding to a sequence of state transitions $\langle s_1, s_2, \dots, s_k \rangle$, such that $s_1 \in \gamma(s_0, a_1)$, $s_2 \in \gamma(s_1, a_2)$, \dots , $s_k \in \gamma(s_{k-1}, a_k)$, and $s_k \in S_g$.

Definition 3. A planning problem \mathcal{P} for a restricted state-transition system $\Sigma = (S, A, \gamma)$ is defined as a triple $\mathcal{P} = (\Sigma, s_0, g)$, where $g \in S_g$.

Definition 4. A solution to a planning problem \mathcal{P} is a sequence of actions $\langle a_1, a_2, \dots, a_k \rangle$, corresponding to a sequences of state transitions $\langle s_1, s_2, \dots, s_k \rangle$, such that $s_1 \in \gamma(s_0, a_1)$, $s_2 \in \gamma(s_1, a_2)$, \dots , $s_k \in \gamma(s_{k-1}, a_k)$, and $s_k \in S_g$.

5.2.3 Characterisation of the refactoring planning problem

In order to select an appropriate planner for the refactoring planning problem it should be characterised as a planning domain. This characterisation also takes into account the restriction we have imposed on the refactoring planning problem in this PhD Thesis Dissertation. According to the basic model defined for automated planning, the refactoring planning system can be represented as depicted in Figure 5.2(b). In the refactoring planning problem:

The system Σ is a software system, more precisely its source code, which can be evolved through source code transformations. The system receives no events, just actions. It remains static and only gets changed through the execution of the transformations performed by the controller.

The controller is a developer operating a source code transformation tool, such as a text editor or an IDE with more advanced source code transformation capabilities.

The planner is a refactoring planner. Since the system Σ is static, the planner works offline –it does not receive nor need information about the current state of the system but only manages the states the system traverses during the plan execution. It can also be an external tool to the system and the controller.

The description of the system is a representation of the software system’s source code, *eg.* AST, together with logic queries that produce additional knowledge about the system, and the specifications of the source code transformations available to the controller, represented in terms of AST transformations.

The initial state of the system is the AST of the current “snapshot” of the software system’s source code.

The objective is to remove a bad smell from the software system or, in more general terms, to apply a complex refactoring sequence. This goal has to be achieved in such a way that the final state reached represents a system whose observable behaviour is equivalent to the initial state’s system.

The plan is a sequence of refactorings and maybe other non-behaviour preserving source code transformations. This sequence of source code changes can be successfully applied over the original system with no precondition-violation errors. The transformation sequence, as a whole, produces a new version of the system that preserves the observable behaviour of the original system.

The actions are refactorings and other source code transformations of different complexity degrees that the developer has to apply to the system with the aid of a programming tool.

The observation is the source code of the software system. This source code can include references to entities for which only their binary code is available. Nevertheless, the observation also contains the basic information about these references to binary entities, such as symbol declarations, signatures, etc.

Determining which restricted model assumptions a planning problem meets is even more important for characterising the problem and choosing a proper planner for it. Those planners which do not support the required extensions can be discarded. As for the refactoring planning problem:

Assumption A0 (finite Σ): Does not hold.

The system can change through an infinite set of states. Adding new entities to the system’s AST, such as adding an unreferenced new class, produces a new state. Adding an infinite number of new entities will produce an infinite number of states. Since the number of distinct entities that can be added is finite, the number of states that can be produced is countable.

The size of the state space can be finite if we restrict the number of actions allowed in a plan. Nevertheless, due to the heuristic nature of refactoring strategies, it is hard to estimate the upper bounds for the plan length. Even with this restriction, the number of refactorings and other transformations applicable to each state is very big, and the combinatorial explosion produces huge state spaces. This implies that, in any case, a planner that can deal with very big search spaces will be needed. As a consequence, we have decided to avoid estimating the upper bound for the plan length and we have considered instead the state space to be infinite, as it actually is.

Assumption A1 (fully observable Σ): Holds.

The system Σ is fully observable. The function η is the identity function. $\forall s \in S; \eta(s) = s$. We have complete knowledge of the current state of the system Σ .

It can be argued that, in some cases, we do not have access to the full system’s source code involved in the refactoring process. In the observed system, there might be references to

entities whose source code cannot be accessed because they are available in an executable format only or are not available at all. However, accessing the source code of these elements is not relevant. Since binary code is not available for being manipulated either, we do not consider these binary elements to be part of the system. The planner has complete access to the system Σ : the source code that can be seen and manipulated by the controller.

Assumption A2 (deterministic Σ): Holds.

Every change applied to a software system's source code by the developer produces one and only one system state. The same transformation performed over the same system's source code always produces the same new version of the system.

Assumption A3 (static Σ): Holds.

The system can only be changed by the actions performed by the developer. It receives no events, therefore this assumption holds.

Assumption A4 (restricted goals): Does not hold.

The goal cannot be simply specified as a set of conditions over a system's state, such as: a state where the targeted bad smell is not present. Not every transformation sequence leading to the bad smell removal is a valid plan, but only those which preserve the observable behaviour of the system. Moreover, we want to formulate goals in terms of requests for executing a certain transformation. Therefore, this assumption does not hold.

Assumption A5 (sequential plans): Holds.

Although the problem can be formulated to deal with the concurrent evolution of a software system, in this Thesis we are not addressing that scenario. Assuming there is only one developer changing the software system at a time, all the modifications are meant to be applied in sequential order, therefore this assumption holds.

Assumption A6 (implicit time): Holds.

Although the process of applying a transformation to the source code of a software system will take some time, this is meaningless for the purpose of planning. Time is not relevant to the refactoring problem. Executing actions can be simplified as an instantaneous process. This assumption holds.

Assumption A7 (offline planning): Holds.

In the typical usage model we expect for our technique, the developer requests a plan and waits for the planner to produce it. The system is not meant to be changed during the planning process and nor do external events exist. The system is static, therefore this assumption holds.

The refactoring planning problem adheres to assumptions, A1, A2, A3, A5, A6 and A7. This has to be taken into account when selecting a proper planning approach for this problem. Any automated planner suitable to be used in the refactoring planning problem should support an extended planning model, with relaxed assumptions for A0 and A4. In addition to this characterisation, the issues previously discussed in Section 4.5, regarding the necessary computation capabilities, representation detail, the ability to use non-deterministic control structures, etc., also have to be taken into account.

As supplementary remarks, we consider the following features to be specially relevant for selecting an appropriate planner:

- Besides the state space being infinite, the size of the state representation of a software system's AST is huge, even for small systems.
- The actions in this planning problem are rather complex. Refactorings are complex transformations that cannot be described with simple lists of positive and negative effects.
- The knowledge about how to remove a bad smell or how to apply a certain complex refactoring process is always being revised, improved and is constantly growing.

Therefore, a planner suitable for the refactoring planning problem should be efficient for big-sized problems, it should also present enough expressiveness to allow complex actions to be specified and it should support the incremental addition of refactoring heuristics.

5.2.4 A variety of planners

There are several planning algorithms and strategies which have been developed during the history of AI planning. Different planning approaches suit different types of problems. Therefore, addressing a particular problem with an automated-planning approach implies identifying the nature of the problem and using a suitable planner. Selecting the most appropriate planner for a problem is central to achieving a successful solution. One useful source to look for planners is the International AI Planning Competition [icab]. This conference is a good reference to find the best planners and planning approaches and select the proper one for a particular problem. This section compiles a brief description of the different families of planning approaches.

Classical planning

Planning for restricted state transition systems is known as classical planning. This family encompasses the most traditional planning approaches and it is only suitable for restricted problems –those fulfilling the eight assumptions previously mentioned in Section 5.2.2. This family of approaches does not deal well with complex and large problems with big state spaces or high combinatorial explosion. Heuristics or search-guiding techniques can be used to improve the efficiency of classical planners but, in general, it is not practical to use them for real problems. Classical planning is also known as STRIPS planning, after the early planner STRIPS [FN71].

Depending on the nature of the search space, classical planning approaches can be classified into state-space and plan-space planning. In **State-Space Planning**, the planning problem is represented so that the nodes in the search space are the states of the state transition system, the edges are the state transitions, and the plan is a path across the search space.

Regarding the search direction, state-space planning can be addressed as forward search or backward search. In **Forward Search**, the initial state for the planner is the current state of the system and the goal state represents the objective to achieve. Planning is performed by applying operators to the initial state until the goal state is reached. On the contrary, in **Backward Search**, the initial state represents the objective to achieve and the goal state is the current state of the system. Searching is performed by selecting operators whose postcondition belongs to the goal state and by reducing them backwards until the current state of the system is reached.

Plan-Space Planning is a completely different approach. The search space is not composed of system states but of partial plans. Each node in the search space is comprised of a set of

operators, and a set of constraints that build up a partially instantiated plan. The edges are plan refinement operations leading to plan completion. These refinements append additional constraints that should be satisfied by the full instantiated plan. The general idea behind this family of planners is to avoid taking decisions that may have to be undone when they are found not to be useful for a valid plan. To do so, they follow the least-commitment principle: constraints are only added when strictly needed. Addition of constraints is delayed as long as possible during the plan refinement process, so less backtracking is needed.

The plans produced by a plan-space planner are composed of a selection of operators, ordering constraints and binding constraints. They may not refer to explicit sequences of actions. A partial plan is comprised of partially instantiated operators, operator ordering constraints, causal links –precedence constraints between preconditions and the actions enabling them–, and binding constraints –constraints restricting the set of valid variable substitutions.

Depending on how the ordering of the operators is given in the resulting plan, planners are also classified as partial-order and total-order planners. Most plan-space planners are **Partial-Order Planners**: planners that produce plans that are partially ordered sequences of operators and are based on least-commitment planning. On the contrary, **Total-Order Planners** are more commonly used for state-space planning. The plans they produce are totally ordered sequences of operators.

The ability of plan-space partial-order planners to generate a plan in an incremental way is an interesting feature for the refactoring planning problem. This enables the generation of incomplete plans which, despite not being executable, can still be useful for the developer. An incomplete plan can represent an overall overview on how the complete plan, or a part of it, would look like and therefore it can still guide the developer. Nevertheless, they cannot be straightforwardly executed and thus, generating incomplete plans is not interesting to our objectives. The computational model of plan-space planning is also a relevant feature for refactoring planning. It allows combinatorial explosion to be handled better than other classical planning approaches. For these reasons, we think plan-space planners are more appropriate for the refactoring planning problem than any other classical planning approach. Unfortunately, the system's state Σ is not transformed during the planning process. All computations are performed over preconditions and postconditions. Therefore, the state of the system at a certain point during the execution of a partial plan is unknown. The unavailability of the current state of the system Σ during the planning process is a major drawback that discourages using plan-space partial-order planners for the refactoring planning problem, as the planner cannot reason about a precise world state and they suffer from lack of expressiveness. The first order logic subset allowed for preconditions and effects is quite restrictive. Therefore, complex predicates to compute metrics, perform structural analysis, represent variable structures, etc. cannot be written due to the lack of the system's current state.

As a summary, among the different classical planning approaches, plan-space partial-order planning seems to be the most adequate for the refactoring planning problem. Unfortunately, like all the classical planning approaches, there are some serious disadvantages, such as their limited expressiveness and their bad efficiency in big search spaces. We have concluded that classical planning approaches are not suitable for the refactoring planning problem, given the characteristics of this domain.

Neoclassical planning

Additional planning techniques were developed on the basis of classical planning. Although these approaches could manage bigger size problems, they are still only applicable to restricted state-transition systems.

Planning-Graph Techniques represented an improvement over classical planning techniques [BF97]. In these approaches, the planning process takes place in two stages and the planner uses a distinctive data structure: the planning graph. In the first stage of the planning process, the planner builds the planning graph and then, in the second stage, it searches for a valid plan as a path across this graph. The planning graph is a layered graph that constitutes a search space different from state spaces and plan spaces. It represents, at the n^{th} level, the predicates that can be reached at the application of the n^{th} action. Layers represent, alternatively, actions and predicates. The graph represents, at each predicate layer, the grounded predicates of the initial state and the ones that can be produced, either by positive or negative effects, by a certain number of actions from the initial state. At each action layer, it represents the applicable actions. It links actions with the precedent predicates that make their preconditions true, and with the successor predicates produced by their effects. For classical restricted problems –those problems holding the eight assumptions of classical planning–, the planning graph can be generated in a very efficient way, and the search of the plan is also performed very efficiently through this reachability graph.

The planning-graph structure represents explicitly all the grounded predicates and all the instantiated actions of a problem. These approaches can be efficient for restricted classical problems, and even for complex ones. Nevertheless, they are not appropriate for the refactoring planning domain. The resulting planning graph for the refactoring planning problem will be unmanageable, given the huge number of predicates needed to represent each system state. Moreover, the refactoring planning system has an infinite number of states. Assumption A0 does not hold due to new terms and symbols that can be introduced in the system. A planning-graph approach cannot fulfill this requirement because representing all the grounded predicates in the graph implies that terms, predicates or symbols cannot be added to the system.

Propositional-Satisfiability Techniques and **Constraint-Satisfaction Techniques** address a planning problem by translating it to other types of problems. This approach works by encoding a planning problem as a propositional formula and evaluating whether this formula is satisfiable or not. A planner uses existing satisfiability decision procedures to find the assignment model that may make the problem formula true. The plan is then obtained from these assignments. We have discarded this family of approaches because we think encoding refactorings and transformations into the needed formalism is not feasible.

The latter approach, constraint-satisfaction techniques, translates a planning problem into a constraint-satisfiability problem (CSP). The problem has to be encoded as a set of variables, a domain and a set of constraints. The plan is searched by computing which values have to be assigned to the variables within the domain so the constraints are met. In order to use a CSP planning approach the problem has to be translated and encoded as a CSP problem too, and therefore, we also think this approach is not feasible for addressing the refactoring problem. CSP is particularly well suited to supporting planning-graph approaches, plan-space techniques and planning with time and resources.

Planning with uncertainty, time and resources

More advanced families of planners allow some of the eight restrictions of classical planning to be relaxed. If the system is only partially observable, and the actions can be non-deterministic, then the system does not fulfill restrictions *A1* and *A2*. Planning for such domains is known as **Planning with Uncertainty**. For some problems, time can be relevant and therefore, has to be represented. For example, actions can have a duration, or goals can contain time constraints. In the real world, a track moving from place *A* to place *B* does not immediately occupy *B* after leaving *A*. If time has to be considered in the planning problem, assumption *A6* has to be removed and the approaches to be used are known as **Planning with Time**. Additionally, representation of resources may be needed. Resources are entities that are created and consumed and whose availability is relevant to the planning problem. **Planning with Resources** deals with the problem of planning with a limited number of resources in a given time. Planning with resources is also called **Scheduling**.

None of these features are needed for the refactoring planning problem. Therefore, we have not considered any of these approaches.

Planning with heuristics and control rules

One of the biggest issues in planning has been combinatorial complexity, even for restricted problems. Most planning approaches are very inefficient or cannot deal with big size problems. This is the case for the refactoring planning problem. As we have already mentioned, state representations are not only huge, they do not meet assumption *A0* either. The refactoring planning system can have an infinite set of states.

According to [AKN09], the best results for this kind of problems have been obtained by using heuristics and rules to guide the planning process and to overcome the difficulty of big search spaces. Heuristics and control rules can be formulated over how to obtain the new nodes to explore (branching), how to select the next nodes to visit (refining) and how to prune the search-space by removing or avoiding useless nodes (pruning). Regarding the relation between the heuristics used and the problem domain, planners can benefit from domain-independent or domain-specific heuristics. The former relate to improving efficiency with the aid of tweaks that are domain-agnostic but specific for a particular planning algorithm or a part of it. The latter compile the problem's domain knowledge into rules that can be used by a planner. Domain-independent heuristics have been useful to greatly improve the efficiency of classical planning approaches. Nevertheless, most modern planning approaches address big size problems by taking advantage of domain-specific heuristics.

How the domain knowledge of a problem is embedded into a planner can also serve to classify and evaluate a planning approach. Regarding the degree of domain-tailoring, planners can be classified as domain-independent, domain-configurable and domain-specific. **Domain-Independent** planners use only the domain knowledge embedded in operator definitions. They do not need additional knowledge to be specified, so they are easier for a non-planner-savvy user to use. Generally, they are less efficient than domain-configurable or domain-specific planners. **Domain-Configurable** planners use a considerable amount of domain knowledge as input, which serves as control rules for the planning process. Their efficiency relies heavily on the domain-knowledge specifications given by the user, but with a small amount of knowledge they can easily outperform domain-independent planners. Nevertheless, their major drawback is that these specifications are difficult to write and to debug, so they are harder to use

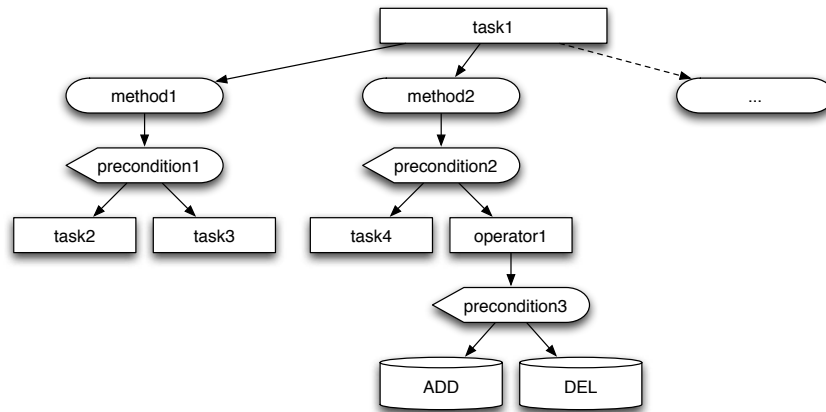


Figure 5.3: *Elements of a hierarchical task network used in HTN planning.*

than domain-independent planners. Domain-configurable planners suit problems for which the domain-knowledge is given in the form of “recipes”. **Domain-Specific** planners are custom-developed from scratch for a particular problem domain, although they can be based on the existing planning approaches. They are useful for problems whose domain knowledge is well-known and constitutes a closed set. The most successful and efficient planners belong to this category. However, they are harder to develop.

5.3 JSHOP2: a hierarchical task network planner

Of all the planning approaches we have analysed, **Hierarchical Task Network (HTN) planning** [GNT04, Chapter 11] provides the best balance between search-based and procedural-based planning approaches for the refactoring planning problem. We have explored other approaches, such as partial-order backwards planning [GNT04, Chapter 5], only to discover that combinatorial explosion and lack of expressivity disallow their application in the refactoring planning domain. For the refactoring planning problem we are using JSHOP2, an HTN forward planner.

5.3.1 HTN planning

HTN planning belongs to the family of modern planning approaches, which use control rules and heuristics extensively in order to support large and complex problem domains. More precisely, HTN planning belongs to domain-configurable planning, and uses mainly domain-specific heuristics, which are focused on branching. These heuristics improve the efficiency of the planning process by guiding how the new nodes to explore are to be obtained. With respect to the search space, HTN planning techniques are similar to state-space planning. The system is represented by sets of terms which constitute states, and the actions of the planner that are specified as state transitions. Regarding the search direction, we can find forward and backward HTN planners. Moreover, HTN planners add an additional search-direction option, they can go up –from operators to goals– or down –from goals to operators. As for the degree of commitment during the plan search, we can also find total-order and partial order planners.

HTN planning introduces the concept of “task”. Tasks are specifications that describe how a problem is solved by decomposing it into smaller problems. They model actions by the composition of simple operators or other tasks. Simple tasks are the typical planning operators, with a precondition and effect lists for added and deleted terms. Composed tasks can be decomposed by different strategies, known as methods, which model the domain by defining which subtasks should be performed to accomplish another one. A method has a precondition and a list of tasks describing a task decomposition. The whole set of tasks defining the domain knowledge of a particular problem comprises a Task Network. The goal of a plan for an HTN planner is not to achieve a state but to execute a task. The actual search for a plan takes place in the non-deterministic selection and instantiation of methods and operators, which is guided by the task network. The basic elements of a hierarchical task network are illustrated in Figure 5.3.

One of the most interesting features of hierarchical task network planning is that the heuristics that guide the planning process are written as “recipes” about how to perform each particular task. Tasks networks comprise “recipes” on how to achieve certain objectives. They allow domain knowledge and heuristics to be included by describing which subtasks should be performed to accomplish another one. This way to specify the domain knowledge is quite similar to how a human thinks about solving a planning problem. Furthermore, HTN planning suits the coexistence of different strategies. Many specifications can be implemented and the planner will search for the ones that apply to each particular case.

HTN planning and forward search allow very expressive domain definitions which can lead to very detailed domains with a lot of domain knowledge. According to [EHN94] HTN planning surpasses classical planning approaches in expressiveness. This implies that HTN can be used in a more complex and broader set of complex planning problems and planning domains than classical planning. Other authors have lowered the augmented expressiveness of HTN planners over classical planners [LN07], and have stated that they are equivalent in terms of which set of problems they can address. Nevertheless, they recognise the relevance of HTN planners in practice. Theoretically, an HTN planner can be translated to an equivalent classical planner, but they conclude that this is not practical. The use of task networks to specify domain knowledge is user-friendly and enables the HTN planning to be used in many more practical cases than classical planning.

HTN forward planning allows for complex computations that are not available for classical planners. In HTN forward planning, tasks are reduced in the same order as the operators will be applied, therefore the current system state is always known. This allows numerical computations to be included that interact with external information sources and deal with imperfect-information systems. For our problem, the ability to perform numerical computations over the current state of the system is crucial to allow for metrics computation. And the chance to interact with external systems permits external functions and procedures to be attached, such as, for example, to support user queries.

HTN planners have also proved to be very efficient and suitable for large size problems. The task networks compiling the problem’s domain knowledge can guide the planning process in a very efficient way because they are used to extensively prune the state space. An HTN planner does not check all the operators applicable to a certain state, but only those explicitly defined in the methods that apply to the task being performed.

In order to apply HTN planning for the refactoring planning problem, we represent the elements of our problem domain as displayed in Table 5.5. We use a logical representation of the current AST of the system as the current state in the planner. Operators are used to represent

World's state:	AST represented by first-order logic predicates
Operators:	atomic changes to the AST (mainly add, delete and replace)
Tasks:	transformation parts refactoring parts non-behaviour preserving transformations refactorings refactoring strategies
Goal:	Executing a smell correction strategy Executing a refactoring application strategy
Planning problem:	Execute a particular refactoring strategy over a particular version of a system

Table 5.5: *Interpretation of the refactoring planning problem as an HTN planning problem.*

atomic transformations –add, delete and replace– over the basic elements of the AST. Refactoring strategies, simpler refactorings and non-behaviour-preserving transformations (NBPT) are implemented as tasks, which can be further decomposed into other tasks. This decomposition construct will allow us to split strategies into simpler ones and to attach preconditions to them. Therefore, this allows dependencies to be specified, and to avoid conflicts between behaviour-preserving transformations or parts of them. Invocation of transformations, refactorings and refactoring strategies will be described as task decompositions. Complex system queries and preconditions are formulated over the logical terms representing the system's AST. This is possible due to the rich expressiveness of the HTN forward planner we use. Depending on the goal defined by a refactoring strategy, the task which implements it can be addressed to the application of a refactoring, the removal of a bad smell, or any subpart of these activities. The domain definition compiled in the task network represents the heuristic empirical knowledge, or “recipes”, of the developer on how to apply these transformations.

The preliminary ideas on using an HTN planning approach to tackle the problem of design smell correction with refactoring plans were presented in [Pér08].

5.3.2 Features of JSHOP2

JSHOP2 is an HTN partial-order forward planner developed by the University of Maryland [IN03, Ilg06, JSHa]. It is a JAVA implementation of the SHOP2 (Simple Hierarchical Ordered Planner) planning algorithm [NAI⁺03], which obtained one of the top prizes in the 2002 International Planning Competition [icaa]. Being a forward planner, the search for a plan is performed in the same order as the plan should be executed. This implies that the current state of the system is available during the planning process, and therefore, this guarantees support for arbitrarily complex queries that can be written in PROLOG style as Horn-clause-like axioms. Being a partial-order planner, the order constraints are delayed as much as possible during the planning process. This allows methods to be decomposed into a partially ordered set of subtasks, and it allows the creation of plans that interleave subtasks from different tasks. According to the authors, JSHOP2 scales well for problems of very large size.

The JSHOP2 planner includes a problem solver that computes system queries in an efficient way. Special queries, named “call terms”, allow JAVA functions, which are external to the planner

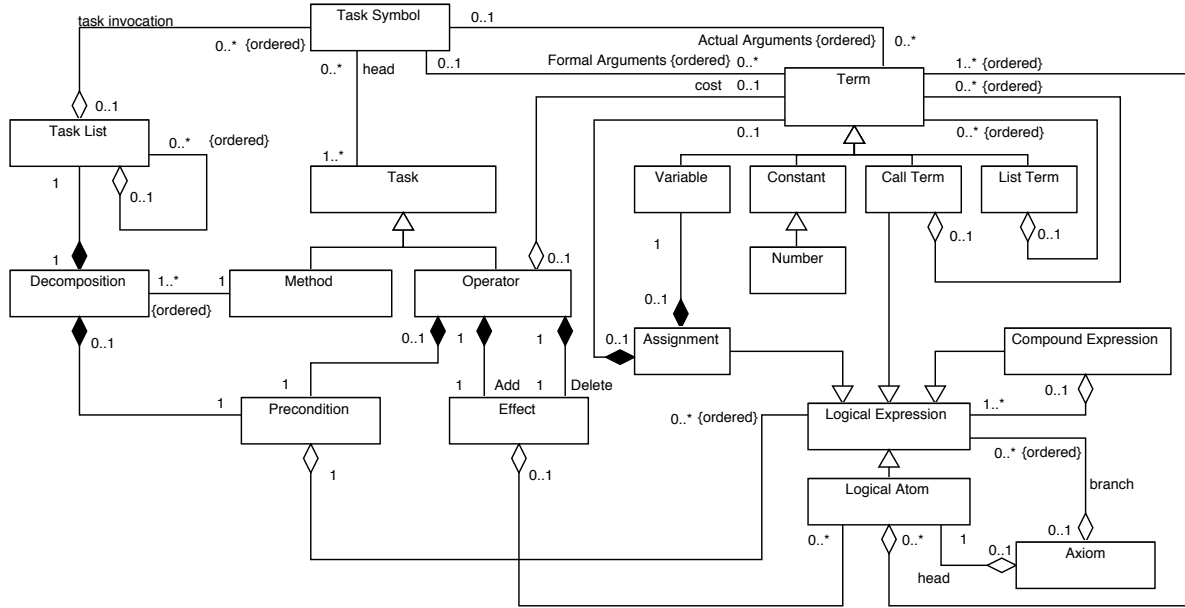


Figure 5.4: *A simplified model of JSHOP2's HTNs*

tool, to be called during the planning process. In order to write problem specifications for the planner, the domain knowledge –the heuristic rules– has to be written as HTNs in a language with LISP-like syntax. The representation of the system's states has to be given to the planner as a set of logical terms in the same LISP-like language. A distinctive feature of JSHOP2 is its precompilation stage. When fed a planning problem, the tool precompiles the problem and produces a JAVA program. This JAVA program contains an optimized version of the problem definition –the rules and the system's initial state– and the planner itself. In order to solve the planning problem, this JAVA code has to be compiled and executed. The 1.0.3 version of the tool [JSHb] is used here.

As previously mentioned, JSHOP2 computes the operators of the plan in the same order as they would be applied. The planner starts from the current state of the world and searches for operators to apply to it in order to reach a set of desired goals. At each search step, it is possible to apply the operator to obtain the new state. Therefore, the complete system's state is available for the planning algorithm at each planning step. This allows more informed heuristics to be used that are more relevant to efficiency than anything else. The most important consequence of this feature is that almost every useful computation can be performed. During the planning process we can use: numerical computations, integration of external tools, calls to external tools, Horn-clause inferencing and very expressive domain representations.

5.3.3 Elements of a JSHOP2 planning problem

We compile here, for reference purposes, the description of the elements of a JSHOP2 planning problem, hence we can later discuss how refactoring strategies are written and computed using this tool.

A simplified version of the HTN elements used in JSHOP2 is depicted as a class diagram in Figure 5.4. **Tasks** can be decomposed into other tasks or **task lists** by different alternative decompositions. Selecting a particular decomposition can depend on the fulfillment of a **precondition**. Two task types can be used: compound tasks and primitive tasks, depending on whether a task has to be decomposed them into simpler ones in order to be achieved or not. Compound tasks are achieved by decomposing it in simpler tasks which are specified with **methods**. These include a precondition, which indicates the conditions under which this decomposition can be attempted, and a task list, which specifies the simpler tasks that should be carried out in order to achieve the compound task. Primitive tasks are achieved by applying **operators**, which describe the actual transformations of the world's state. The transformation is specified by indicating which terms the operator will **add** to and **delete** from the current world's state. The application of an operator can also be restricted with a precondition. First order logical predicates are used to query the system's current state and to call external procedures. **Logical expressions** and **logical atoms** allows additional knowledge to be queried and derived from the world's state. **Call terms** allow functions, external to the tool, to be called during the planning process.

The planning process starts with a task network, which is structured with operators and methods, a set of ground predicates that form the system's initial state, and a goal, represented by a grounded task list. The planner searches for methods and operators to achieve the goal tasks and the pending tasks that result from the consequent decompositions. If a task can be solved with an operator, the planner applies the operator, if the task is solved with a method, it is substituted by the task list defined in the method's decomposition. Some considerations are worth noting. Task lists, or task invocations can only appear within a method's task list decomposition, while queries and variable bounding only take place within the method's and operators' precondition. The changes to be performed to the system's state are only specified as operators' effects lists, therefore the system state can only be modified by applying an operator.

A detailed review of the different elements of a JSHOP2 planning problem follows. This review uses and complements the models depicted in Figures 5.4 and 5.5. Using these models, it will be easier to document in this chapter how refactoring strategies are implemented and supported with JSHOP2. This will be done by matching these JSHOP2 models with the refactoring strategy models presented in Chapter 4 (See Figures 4.4 and 4.5). For reference purposes, the description of the subset of the JSHOP2 language used here, including syntax examples, is compiled in Appendix A. This reference has been extracted from its original source [Ilg06], and only includes the subset of the language we have actually used. For the complete language, the original source should be consulted [Ilg06].

Terms

Terms are the most basic entities in the language. JSHOP2 defines simple terms such as: numerical constants, and variable and constant symbols; and more complex terms such as: list terms, which are ordered sets of terms, and call terms, which represent external procedure calls. When a call term is evaluated, an external JAVA procedure is executed. After being evaluated, the call term is substituted by the resulting term produced by the external procedure. An ordered set of terms can be passed to call terms as their arguments. JSHOP2 includes some built-in call procedures such as comparison functions. The planner supports user-defined procedures, which have to be implemented in JAVA.

A term t is:

- a numerical constant, variable, constant;
- a list of terms: $t : \vec{t}$, where $\vec{t} = (t_1 \ t_2 \ \dots \ t_n)$ and each t_i is a term of the list;
- a call term: $t : ct(\vec{t})$, where ct represents the external procedure and \vec{t} represent its arguments, which are also terms.

Logical atoms and logical expressions

Logical atoms and logical expressions are affirmations about the state of the world that can be evaluated during the planning process. A logical expression is any expression that can be evaluated to a boolean value. Logical atoms are first-order-logic predicates that will be evaluated to `true` if they exist in the current state of the world or can be derived from it and to `false` otherwise. Call terms can also be used as logical expressions. Depending on their usage context *-i.e.* when they appear within a precondition-, call term results can be interpreted as boolean values. Empty results are interpreted as `false` and non-empty results as `true`. Logical expressions can be composed in JSHOP2 with several boolean operators: `and`, `or`, `not`, `implies` and `forall`. Assignments are also available as special kinds of expressions that force a term to be bound to a variable. A logical atom is said to be grounded when it contains no variables or, at least, no unbound variables.

A logical atom a is an expression:

$$a = p(\vec{t})$$

where the constant p designates the predicate symbol and \vec{t} represents the predicate's terms.

A logical expression e is either:

- a logical atom a ;
- a call term ct ;
- or an expression $e = op(\vec{e})$, where the function op designates a logical operator and \vec{e} represents a list of logical expressions.

Logical preconditions

Logical preconditions are logical expressions used to control whether a particular method or operator can be included in a plan or not. During the planning process the method's and the operator's preconditions are evaluated. The planner searches for those methods or operators whose precondition holds in order to add them to the plan.

Preconditions are not only meant for selecting methods and operators. The evaluation of a precondition is also used to perform certain computations. They are useful for querying the system with axioms or system-state predicates in order to assign values to unbound variables. In the case of methods, preconditions allow any necessary information lacking to be found before invoking tasks, so this information queried can be used as tasks' arguments. In the case of operators, it allows any unbound variables to be bound, and therefore grounded system-predicates to be produced to apply the operators' add and delete lists of effects to the system's state.

Regular preconditions are composed of any kind of logical expression. Additionally, JSHOP2 offers two special precondition types that can be used for optimizing and control the planning process: the first satisfier precondition, and the sorted precondition. First satisfier preconditions are useful to optimize the computation of plans. This type of precondition will be evaluated until a valid substitution is found or until none can be found. No other valid substitution will be tried, even if the first one does not lead to a valid plan. They can be used to prune the state space extensively and to avoid searching along useless paths.

Sorted preconditions can be used to specify the order in which the possible variable substitutions will be attempted when computing the precondition. They include the evaluation of a comparison function. When searching for satisfiers for the precondition, the planner will order the valid substitution sets according to the result of the comparison function. Sorted preconditions may serve to define priorities for the available search paths.

A logical precondition P is either:

- a logical expression e : $P = e$;
- a first satisfier precondition: $P = first(P')$;
- or a sorted precondition: $P = sorted(P', v, comp)$, where P' is a precondition, v is the variable upon which the binding attempts for P' should be ordered and $comp$ is the comparison function used for ordering.

Axioms

Axioms are Horn-clause-like expressions that can be used to derive additional knowledge from the system's state. An axiom definition is composed of a head and an ordered set of branches. The axiom's head represents an affirmation on the current state of the system that does not appear in the set of predicates which define the state. Branches represent different ways to compute the axiom's head or to search for its validity. The branches of an axiom are ordered, therefore a branch will be evaluated and a unification attempt will be carried out only if all the preceding branches have failed. Axiom heads are logic atoms and thus logic expressions. They can be used as if they were regular system state predicates. An axiom is "computed" by using its head predicate, for instance, within a precondition.

An axiom A is a definition:

$$A = \{h, \vec{B}\}$$

where:

h is a logic atom $p(\vec{t})$ that defines the axiom as the axiom's head and

\vec{B} is a list of axiom branches $(B_1 B_2 \dots B_n)$, each B_i being a logical precondition P_i ,

thus:

$$A = \{p(\vec{t}), \vec{P}\}$$

Tasks, task lists and task invocations

Tasks are the fundamental abstract constructs used in HTN planning to define and organise the planning problem's domain knowledge and the actions available to the controller. As previously mentioned, a task is a structured specification of how to achieve a certain goal. This specification

is based on the decomposition of tasks into more simple ones. The organisation of a planning problem's domain knowledge into sets of these task decompositions builds up hierarchical task networks.

A task is identified by its name and its number of arguments, which, in the case of JSHOP2, are given as an ordered sequence of terms. Tasks sharing the same name and number of arguments represent different procedures to achieve the same task. A task name and arguments constitute a task symbol that can be used as a task head in order to define a task, or as a task invocation in order to reference the task. Selecting and applying, among the available versions of the same task, those which lead to a valid plan is one of the decisions taken by a planner during the planning process. Depending on the task complexity, their specifications can either be written as operators –primitive tasks– or as methods –compound tasks. Primitive tasks can simply be performed by applying an operator, while compound tasks have to be described with methods that specify how the more complex task can be achieved by decomposing it into simpler ones.

To better describe how tasks are used in JSHOP2, in our model, we have distinguished between task definitions, or simply tasks, and task symbols. Task invocations are those references to task symbols that appear within a method's task decomposition task list. In JSHOP2, planning is performed in the same order as the operators are applied to the system's state; therefore, the sequence of simpler tasks needed to perform a more complex one has to be given as ordered sequences. These ordered sets of task invocations are referred as task lists. They are used within methods to describe how a more complex task should be decomposed or within a planning problem definition to specify a planning goal.

A task symbol T is an expression:

$$T = h(\vec{t})$$

where:

- h is a constant symbol that represents the task and
- \vec{t} is a list of terms that represent the task arguments.

Depending on the type of task, T can be a primitive task symbol or a non-primitive (or compound) task symbol.

A task list \vec{T} is an ordered list:

$$\vec{T} = (T_1 \ T_2 \ \dots \ T_n)$$

where each T_i is a task symbol.

Operators

Operators, or primitive tasks, are atomic specifications of how to achieve a task. They are applied to a system in order to change its state. Operators are composed of a precondition, a list of negative effects, a list of positive effects and the optional cost of applying the operator. The precondition specifies the conditions under which the operator can be applied. As mentioned before, the precondition can also be used to complete the information needed to apply the operator effects over the current system's state. This can be used, for example, to bind any remaining unbound variables prior to the application of the operator effects. The operator effects are expressed as two sets of logical atoms. More precisely, these logical atoms are restricted to being grounded system-state predicates. They can be defined with variables but, at least at their application time, they have to be instantiated into grounded predicates. The operator lists of

effects define how the operator changes the system's state. If the planner selects and instantiates an operator for including it in a plan, it applies the operator to the system's current state by removing and adding the logical terms included in the negative and positive effects lists.

More formally, we can describe an operator O as a set:

$$O = \{T, P, D, A\}$$

where:

- T is a primitive task symbol $h(\vec{t})$, a primitive task h with an ordered list of parameters \vec{t} ;
- P represents the operator's precondition;
- D is a set of logic atoms $\{d_1 d_2 \dots d_n\}$ that represents the operator's negative effects and
- A is a set of logic atoms $\{a_1 a_2 \dots a_m\}$ that represents the operator's positive effects list.

In JSHOP2 HTNs, effects can also be specified with a special construct that allows a collection of effects to be added or removed with a single operator. More precisely, "forall" effects allow all predicates holding a certain condition to be added or removed.

A "forall" effect list A or D is specified as a special effect list:

$$F = \{V, C, E\}$$

where:

- V is a set of variables;
- C is a logical expression that represents the "forall" condition and
- E is a set of logic atoms that represent the system state predicates meant as operator effects.

When this effect list is applied during the planning process, all the possible effects in E that are produced by substituting the variables V with the different values that should make C true are either removed or added (depending on the effect list, D or A , where the "forall" effect is used).

Methods

Methods, or compound tasks, are specifications of how to achieve a task by decomposing it into simpler tasks. If the method's precondition is held in the current state of the system, the task can be achieved by performing the list of tasks defined in the method. Methods are comprised of an ordered set of decompositions, which are used to represent alternative ways to perform the method that have to be attempted in a given order. A method decomposition is defined with a precondition and a task list. The precondition describes the conditions under which the decomposition can be attempted. The task list specifies the sequence of simpler tasks that should be performed in order to apply the method's associated compound task. These alternative decompositions within a method definition are visited by the planner in sequential order and their preconditions are considered in the given order. A decomposition is only attempted if the precondition of all the preceding decompositions are not met. If a decomposition's precondition holds, the next decomposition is not attempted, even if the former decomposition's task list does not lead to a valid plan.

More formally, we can describe a method M as a set:

$$M = \{h(\vec{t}), (P_1\vec{T}_1 P_2\vec{T}_2 \dots P_n\vec{T}_n)\}$$

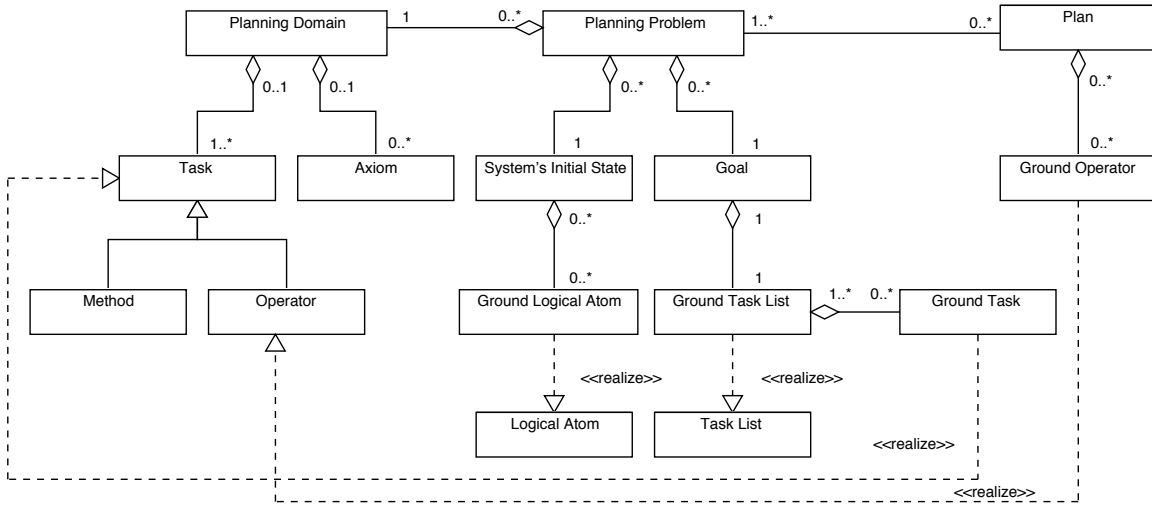


Figure 5.5: A model for HTN planning problems

where:

$h(t)$ represents a compound task with an ordered list of parameters \vec{t} and
each $P_i T_i$ represents an alternative decomposition, with P_i being the precondition and \vec{T}_i the task list of decomposition i .

Planning domains, planning problems and plans

A planning domain is used to compile and specify all the domain knowledge available for a given problem. In JSHOP2, a planning domain is comprised of a task network, which includes operators, methods, axioms, which additional knowledge to be derived from the system's state, and external procedures (see Figure 5.5). A planning problem is defined by a planning domain, the initial state of the system and the goal to achieve. The system state is represented with a set of ground logical atoms. In HTN planning, the goal is given as a grounded task list. Finally, a plan produced by the planner for a given planning problem is a sequence of grounded operators. For each planning problem, if the planner succeeds, it could be able to produce more than one valid plan. In JSHOP2, it is possible to formulate and run several planning problems over a shared domain definition with a single construct and file.

A planning domain is a set:

$$Domain = \{O, M, A, P\}$$

where:

- O is a set of operators;
- M is a set of methods;
- A is a set of axioms and
- P is a set of external procedures.

A planning problem is a set:

$$Problem = \{Domain, S, G\}$$

where:

Domain is a planning domain;

S is a set of ground system state predicates $\{a_1, a_2, \dots, a_n\}$ and

G is a list of ground task symbols \vec{T} .

5.3.4 Managing variables and variable scopes

Before describing how refactoring strategies are written as HTNs in JSHOP2, some additional considerations regarding variables and variable scopes should be addressed. We have introduced in the planner the concept of “persistent variables” to manage “variables” in certain situations. Some auxiliary queries and operators have been included in the refactoring planning domain as well, for dealing with persistent variables. These auxiliary elements, or others similar to them, should probably be required in any other complex planning domain.

In JSHOP2 HTNs, the scope of variables is restricted to the method or operator where they appear. Moreover, the scope of variable bindings is also restricted to the method or operator in which the binding is performed. Any logical expression may produce variable bindings, but any variable binding taking place within an operator or a method precondition is not propagated outside the compound task. A variable passed as an argument to a method or operator, and bound within this task scope, does not propagate the binding outside this scope. We can say that variables are passed to methods and operators by value.

Methods and operators can be mistaken for functions or procedures that can have return values or output arguments. In the HTN planning approach we are using, only logical atoms that invoke axioms, or that evaluate against system state predicates, can have variables serving as “output arguments”. Thus, contrary to what can be intuitively thought, methods and operators do not “work” as functions or procedures and do not have return values or output arguments.

Nevertheless, in some cases, it will be needed to access a variable value within a different scope to that where the value was bound to the variable. In order to make variable bindings persist outside of the method’s and operator’s scope we have to “commit” these bindings to a more “permanent storage”. We use the current system’s state as this “permanent storage”. We will refer to these “permanent” variables as persistent variables.

Persistent variables

A persistent variable V_p is a bound variable whose value has been committed to the system’s current state. It is represented with a logical atom of the form:

$$pv(v_p V)$$

where:

pv represents a persistent variable predicate;

v_p is a symbol used to identify the variable and

V is a constant, a bound variable whose value we want to preserve or a ground-bound term.

Querying persistent variables

In order to access a persistent variable v_p from an outer scope, we will use a query Q of the form:

$$Q = pv(v_p V)$$

where:

- v_p is a symbol used to identify the variable and
- V is an unbound variable.

After the evaluation of the persistent variable predicate pv , the value of the variable v_p , whose binding scope was restricted to a method or operator T , will be bound to the variable V , whose scope is independent of T .

Committing persistent variables

In order to commit the value of a persistent variable, we use an auxiliary operator that takes the variable's name and bound value as its arguments. The operator's precondition is trivially true, and its effects lists are meant for adding the corresponding persistent variable predicate to the system's current state.

More formally, an auxiliary task to commit the value of a persistent variable is an operator O of the form:

$$O = \{cpv(\vec{v}), P, D, A\}$$

where:

- cpv represents the “commit persistent variable” operator;
- \vec{v} contains two arguments V_p and V_v for the variable name and the variable value respectively;
- P is trivially true;
- D is the empty list and
- A contains, as a single logic atom, the persistent variable predicate $pv(V_p V_v)$,

thus:

$$O = \{cpv(V_p V_v), true, (), (pv(V_p V_v))\}$$

Removing persistent variables

Adding persistent values also raises the necessity to “clean them up”, when they are no longer needed. We define similar operators that remove these variables from the system's state, in order to avoid increasing the size of the current system's state unnecessarily, or to avoid conflicts due to committing persistent variables with the same name but different values.

More formally, an auxiliary task to remove the value of a persistent variable is an operator O of the form:

$$O = \{dpv(\vec{v}), P, D, A\}$$

where:

- dpv represents the “delete persistent variable” operator;
- \vec{v} contains two arguments V_p and V_v for the variable name and the variable value respectively;
- P is trivially true;
- D contains, as a single logic atom, the persistent variable predicate $pv(V_p V_v)$ and
- A is the empty list,

thus:

$$O = \{dpv(V_p V_v), \text{true}, (pv(V_p V_v)), ()\}$$

Notice that, even if the persistent variable does not exist in the current system's state the attempt to apply the operator during the planning process will not fail due to the precondition being trivially true.

Removing unique persistent variables

In the case, there is only one variable-value pair in the system, we do not need to specify the persistent variable value, so another operator can be used.

An auxiliary task to remove the value of a persistent variable is an operator O of the form:

$$O = \{dpv(\vec{v}), P, D, A\}$$

where:

- h represents the “delete persistent variable” operator;
- \vec{v} contains a single argument V_p for the variable name;
- P contains a logical query $pv(V_p V_v)$;
- D contains, as a single logic atom, the persistent variable predicate $pv(V_p V_v)$ and
- A is the empty list,

thus:

$$O = \{dpv(V_p), (pv(V_p V_v)), (pv(V_p V_v)), ()\}$$

Notice that if the persistent variable predicate is not present in the system's current state, the operator's precondition will fail, the operator could not be applied and the planning process may fail.

Removing all persistent variables

In the case where there are multiple persistent variable predicates in the system's state, with the same name but different values, a different task can be used to remove all of them with a single operator invocation.

An auxiliary task to remove all persistent variables sharing the same name is an operator O of the form:

$$O = \{dapv(\vec{v}), P, D, A\},$$

where:

- $dapv$ represents the “delete all persistent variables” operator;
- \vec{v} contains a single argument V_p for the variables shared name;
- P is trivially true;
- D contains, as a single logic atom, a forall effect $F = \{(V_v), pv(V_p V_v), (pv(V_p V_v))\}$ and
- A is the empty list,

thus:

$$O = \{dapv(V_p), \text{true}, \{(V_v), pv(V_p V_v), (pv(V_p V_v))\}, ()\}$$

The attempt to apply the operator during the planning process will not fail due to the precondition being trivially true. This “multi-remove” operator can be used in the following two cases: when we believe there is only one persistent variable with the given name, when we do

not want the operator to fail, even if the corresponding persistent variable predicate cannot be found in the system's state.

Querying persistent variables safely

In order to check and query a persistent variable we can simply use, as previously mentioned, the persistent variable predicate: $pv(V_p \ v)$ within a precondition. Nevertheless, the precondition, and therefore the planning process, may undesirably fail if the persistent variable does not exist in the system's state when queried.

For querying the persistent variable V_p so that the query never fails, even if the variable is not present in the system's state, we use an axiom A of the form:

$$A = \{qpv(\vec{v}), (B_1 B_2)\}$$

where:

- qpv represents an axiom that queries for a persistent variable;
- \vec{v} contains two arguments V_p and V_v for the variable name and the variable value respectively;
- B_1 contains, as a single logic atom the persistent variable predicate $pv(V_p \ V_v)$ and
- B_2 is trivially true,

thus:

$$A = \{qpv(V_p \ V_v), ((pv(V_p \ V_v)) \ true) \}$$

Querying this axiom will always succeed. If the persistent variable exists, it will return V_p and V_v bound, and otherwise, any unbound variables will remain that way.

5.4 Refactoring strategies as HTNs

This section presents how refactoring strategies can be specified as JSHOP2 task networks and how the refactoring planning problem can be addressed as a JSHOP2 planning problem.

Let us present some general ideas first. Refactoring strategies are implemented as **tasks**, which can be further decomposed into other tasks. This decomposition construct will allow us to split strategies into simpler ones and to attach preconditions to them. Therefore, this allows the specification of dependencies and conflicts between behaviour-preserving transformations or parts of them. Depending on the goal defined by a refactoring strategy, the task which implements it can be addressed to the application of a refactoring, the introduction of a design pattern, the removal of a bad smell, or any subpart of these activities.

Operators are used to implement simple and composed transformations, and can also be guarded by a precondition. The **add** and **delete** lists, which describe the changes that are performed through the application of an operator, are used to represent transformations over the basic elements of the AST. These system elements are specified as first-order-logic predicates by means of logical atoms. Some code queries are implemented with axioms and **logical expressions**. Complex queries, or those which can be hard to derive by logic inference, can be computed by means of external calls, using **call terms**.

The detailed description of our approach about how refactoring planning problems can be implemented with JSHOP2 follows.

5.4.1 System elements

Primitive system elements –AST entities and relations– are represented as first-order-logic predicates, with JSHOP2's logical atoms.

A system element E is specified as a logical atom of the form:

$$E = p(t_1 t_2 \dots t_n)$$

where p is a predicate symbol matching the element type –*i.e.* class, method, call, etc.– and each t_i represents the element properties. Among these element properties, the predicate may include the element's name, a unique identifier, references to other system elements, such as their parent or children within the AST, and any additional element property needed to represent it.

5.4.2 System queries

A system query Q is a logical atom that has to be validated or computed over the current system's state, the available set of axioms and the defined external procedures. A system query is invoked as a first-order logic predicate of the form:

$$Q = q(\vec{t})$$

where:

q represents the query and
 \vec{t} is a list of terms that represents the query arguments.

Query arguments that are constants, grounded terms, or variables which have already been bound to ground terms, are input arguments. This means that they may be used to compute the query, but they are kept unchanged after that. Those arguments which are unbound variables can be used as output arguments. They can be bound after the query has been computed.

The most basic system query, is a query of the form:

$$Q = p(\vec{t}) = E$$

where:

p represents a system element predicate and
 \vec{t} represents the system element properties.

If all the terms in \vec{t} are grounded terms, or variables bound to grounded terms, the query invocation results in the planner searching for the system element E in the current system's state.

If any term in \vec{t} is an unbound variable, the query invocation results in the planner searching for a valid binding for all the unbound variables, so E matches some system element in the current system's state.

Structural queries

A structural system query Q is specified with an axiom of the form:

$$Q = \{q(\vec{t}), (B_1 B_2 \dots B_n)\}$$

where:

q is a predicate that represents the query;

\vec{t} is a list of variables that represent the query arguments and each B_i represents a query branch.

Query branches represent alternative ways of computing the query, and are evaluated in an if-then-elseif-else style. The branch i will only be evaluated if all the preceding branches have failed.

Lexical queries

In order to perform lexical queries, we have to analyse the symbol's substrings. This cannot be done by logic inference, but can be performed using external procedures.

A lexical query Q is specified with a call term of the form:

$$Q = c(\vec{t}) : T$$

where:

c represents the call term;

T represents the term's evaluation result, as returned by the external procedure and

\vec{t} represents the procedure arguments, which should be grounded.

Boolean condition

A boolean condition C is either a query predicate Q , or a composed condition of the form:

$$C = op(\vec{q})$$

where:

op is a boolean operator and

\vec{q} is a set of boolean conditions.

5.4.3 Non-behaviour-preserving transformation steps

A transformation step is either a composed system change, a basic system change or the application of a non-behaviour-preserving transformation. Each of these will be described separately.

System changes

The definition of the basic system changes, which will allow basic system elements to be added, deleted or replaced, is carried out with three basic operators. An invocation of these basic transformations is performed within a task list, in a method's decomposition, as a task invocation of the form:

$$h(\vec{t})$$

where:

h represents the proper basic change, which is a primitive task and

\vec{t} is either a single term $-E-$ in the case of the add and delete transformations, or two terms $-E_1, E_2-$ in the case of the replace operator. These should be ground terms.

Adding a basic system element

The definition of a basic transformation, used for adding a basic system element E , is written as an operator:

$$O = \{add(\vec{t}), P, D, A\}$$

where:

- add represents a primitive task;
- \vec{t} contains a single variable V ;
- P is trivially true;
- D is an empty list and
- A contains solely the variable V ,

thus:

$$O = \{add(V), true, (), (V)\}$$

where the variable V represents the system element to be added to the system's current state.

The element to be added should be passed to the operator when invoking it. At planning time V should be bound to a valid grounded predicate representing a system element.

Removing a basic system element

The definition of a basic transformation, used to remove a basic system element E , is written as an operator:

$$O = \{del(\vec{t}), P, D, A\}$$

where:

- del represents a primitive task;
- \vec{t} contains a single variable V ;
- P is trivially true;
- D contains solely the variable V and
- A is an empty list,

thus:

$$O = \{del(V), true, (V), ()\}$$

where the variable V represents the system element to be removed from the system's current state.

The element to be removed should be passed to the operator when invoking it. At planning time V should be bound to a valid grounded predicate representing a system element.

Replacing a basic system element

The definition of a basic transformation, used to replace a basic system element E_1 for another basic system element E_2 , is written as an operator:

$$O = \{rep(\vec{t}), P, D, A\}$$

where:

- rep represents a primitive task named "replace";

\vec{t} contains two variables V_1 and V_2 ;
 P is trivially true;
 D contains solely the variable V_1 and
 A contains solely the variable V_2 ,

thus:

$$O = \{rep(V_1V_2), true, (V_1), (V_2)\}$$

where:

V_1 represents the system element to be removed from the system's current state and
 V_2 represents the system element to be added to the system's current state.

The element to be added and removed should be passed to the operator when invoking it. At planning time, V_1 and V_2 should be bound to valid grounded predicates representing system elements.

Alternatives

An alternative is specified as a method M . In all cases, the method M representing the alternative can be invoked within a task list as a task invocation with the form:

$$alt(\vec{t})$$

where:

alt represents the alternative's method and
 \vec{t} is a list of variables representing all the parameters needed by the alternative.

if-then alternative

For an alternative of the form:

$$if\ C\ then\ S\ end,$$

where:

C represents a boolean condition and
 S represents a sequence of non-behaviour-preserving transformation steps,

the method M has the following form:

$$M = \{alt(\vec{t}), (P_1T_1\ P_2T_2)\}$$

where:

alt is a compound task that represents the alternative;
 \vec{t} is a list of arguments;
 P_1 represents the condition C ;
 T_1 represents the sequence of steps S ;
 P_2 is trivially true and
 T_2 is the empty list,

thus:

$$M = \{alt(\vec{t}), (C\ S\ true\ ())\}$$

if-then-else alternative

For an alternative of the form:

if C then S₁ else S₂ end,

where:

C represents a boolean condition and

S₁ and *S₂* represent sequences of non-behaviour-preserving transformation steps,

the method *M* has the following form:

$$M = \{alt(\vec{t}), (P_1 T_1 \ P_2 T_2)\}$$

where:

alt is a compound task that represents the alternative;

\vec{t} is a list of arguments;

P₁ represents the condition *C*;

T₁ represents the sequence of steps *S₁*;

P₂ is trivially true and

T₂ represents the sequence of steps *S₂*,

thus:

$$M = \{alt(\vec{t}), (C \ S_1 \ true \ S_2)\}$$
if-then-elseif alternative

For an alternative of the form:

if C₁ then S₁ elseif C₂ then S₂ elseif C₃ then S₃ ... else S_n end

with *n* conditional branches, where:

C_i represents boolean conditions and

S_i represent sequences of non-behaviour-preserving transformation steps,

the method *M* has the following form:

$$M = \{alt(\vec{t}), (P_1 T_1 \ P_2 T_2 \ \dots \ P_n T_n)\}$$

where:

alt is a compound task that represents the alternative;

\vec{t} is a list of arguments;

P_i represents the condition *C_i*;

T_i represents the sequence of steps *S_i*;

P_n is trivially true and

T_n represents the last sequence of steps *S_n*, or the empty list if the *else* part is not present,

thus:

$$M = \{alt(\vec{t}), (C_1 \ S_1 \ C_2 \ S_2 \ \dots \ C_{n-1} \ S_{n-1} \ true \ S_n)\}$$
Loops

Using JSHOP2 HTN elements, loops can be specified with a method whose task decomposition list contains, as its last task, a recursive invocation of the method itself. During the planning

process, the planner will keep invoking the method while the condition holds. This will result in a loop iteration being performed at each recursive method invocation.

The loop condition is specified as the method's precondition. Each different substitution for the variables in the precondition will produce a different iteration. In the case of deterministic loops, those used in refactorings and NBPTs, the method's precondition should be a first satisfier precondition, so the planner will not attempt to "reorder" the loop iterations. Each iteration must succeed independently of their ordering. The planner will only attempt a valid substitution for each iteration. If a valid substitution, which makes the loop's precondition true, leads to a task decomposition that fails, the planner will not try an alternative substitution and the loop planning will fail. This implies that alternative iteration orderings will not be tried because all orderings are supposed to succeed in deterministic loops.

A method decomposition only succeeds if a valid plan can be found for the task list of this decomposition. However, the loop should exit, after its last iteration, when the precondition of one of these recursive method invocations fails. In order to exit the loop while allowing the last iteration's method invocation to succeed, an alternative decomposition is added to the loop's method. This decomposition is attempted only after the last iteration precondition fails and it is composed of a trivially true precondition and an empty task list. This decomposition represents a kind of "exit" branch for the loop method.

A loop is specified with a method M . We shall now describe more formally how different loop types can be represented with JSHOP2 HTNs.

while loops

For a loop of the form:

while C loop S

where:

C represents a condition that has to be true in order to enter the current loop iteration and
 S represents a sequence of non-behaviour-preserving transformation steps,

the loop is represented with a method M of the form:

$$M = \{l(\vec{t}), (P_1 T_1 \ P_2 T_2)\}$$

where:

l is a compound task that represents the loop;
 \vec{t} represents a list of arguments passed to the loop method;
 P_1 is a first satisfier precondition that represents the condition C ;
 T_1 represents the sequence of steps S and contains a task invocation $l(\vec{t})$ as its last task;
 P_2 is trivially true and
 T_2 is the empty list,

thus:

$$M = \{l(\vec{t}), (first(C)(S \ l(\vec{t})) \ true \ ())\}$$

With this loop specification, $P_1 \ T_1$ represents the loop's iteration and $P_2 \ T_2$ represents the exit branch.

while loops with persistent variables

It should be noticed that only the bound values of those variables in \vec{t} are passed from outside the loop, and from one iteration to another. When defining the loop, this list of arguments should be restricted to those variables whose binding is, or should be, performed before invoking the loop, and whose value needs to be used in the loop. If the value of a variable is bound within a loop and has to be kept between loop iterations, the variable should be committed to the system's current state as a persistent variable.

When using persistent variables, they should be committed with the first task invocations within T_1 . Their value should be gathered within the first queries to be evaluated within P_1 , therefore:

$$P_1 = \text{first}((qp_v(V_{p1}V_{v1})) (qp_v(V_{p2}V_{v2})) \dots (qp_v(V_{pn}V_{vn})) C)$$

where:

qp_v are persistent variable queries;

V_{pi} are the committed variable names and

V_{vi} will be bound to the variable values if they have already been committed, or otherwise, remain unbound until the binding is performed in C ,

thus:

$$T_1 = (cp_v(V_{p1}V_{v1}) cp_v(V_{p2}V_{v2}) \dots cp_v(V_{pn}V_{vn}) S l(\vec{t}))$$

In order to access the persistent variables within C and S , V_{vi} should be used.

Any variable whose value has been bound within the loop, and should not be needed outside the loop, should be removed when exiting the loop. Therefore, the task list T_2 should contain task invocations for the proper auxiliary operators that will remove these variables:

$$T_2 = (dp_v(V_{p1}V_{v1}) dp_v(V_{p2}V_{v2}) \dots dp_v(V_{pn}V_{vn}))$$

or

$$T_2 = (dap_v(V_{p1}) dap_v(V_{p2}) \dots dap_v(V_{pn}))$$

do-while loops

In order to represent “do-while” loops, their representation is split into two methods. To define the conditions under which an operator or a method can be applied, preconditions, which are evaluated before operator's effects or method's task list decomposition, have to be used. For “do-while” loops, the condition is evaluated after the first iteration, which is always executed, and it is evaluated to control entering the next loop's iteration. The loop's body has to be executed at least once, and then the next iterations can be tackled as in a “while” loop. Therefore, we will use a method for representing the first iteration and for invoking a second method where we place the loop's condition and, again, the iteration body.

More formally, for a loop of the form:

$$\text{loop } S \text{ while } C$$

where:

S represents the sequence of transformation steps comprising the loop's iteration body and

C represents a condition that has to be true in order to enter the next loop iteration,

the loop is represented with two methods M and M' of the form:

$$\begin{aligned} M &= \{l(\vec{t}), (P\ T)\} \\ M' &= \{l'(\vec{t}), (P_1T_1\ P_2T_2)\} \end{aligned}$$

where:

- l is a compound task that represents the loop and that should be used for invoking the loop;
- \vec{t} represents the list of parameters passed to the loop method;
- P is a first satisfier precondition containing all the queries Q needed by the loop's first iteration, but it should be evaluated to true independently from the loop's exit condition C ;
- T represents the sequence of steps S and contains a task invocation $l'(\vec{t})$ as its last task;
- l' is a compound task that represents the loop's condition and next iteration and that should be invoked just from l ;
- P_1 is a first satisfier precondition that represents the condition C ;
- T_1 represents the sequence of steps S and contains a task invocation $l'(\vec{t})$ as its last task;
- P_2 is trivially true and
- T_2 is the empty list,

thus:

$$\begin{aligned} M &= \{l(\vec{t}), (first(Q)\ (S\ l'(\vec{t})))\} \\ M' &= \{l'(\vec{t}), (first(QC)\ (S\ l'(\vec{t})\ true\ ()))\} \end{aligned}$$

With this loop specification, $P\ T$ represents the loop's first iteration, $P_1\ T_1$ represents the loop's next iterations and $P_2\ T_2$ represents the exit branch.

for-each loops with explicit collections

As mentioned in Section 4.1.2, iterative procedures are frequently due to the need to apply a transformation of executing a task for each item holding a certain property. In these cases, “for-each” loops are useful. In imperative programming languages, this type of loops are typically specified with a collection of items, and a variable to hold an item at each iteration. In order to represent “for-each” loops with JSHOP2 HTNs, we use a single method, similar to those already described for other types of loops, and some queries, which are required to iterate over the collection.

More formally, for a loop of the form:

foreach I in C loop S

where:

- I is the auxiliary variable used in the loop's iteration as the loop's variant and for storing each item of C ;
- C is a collection the loop iterates over and
- S represents a sequence of non-behaviour-preserving transformation steps,

the loop is represented with a method M of the form:

$$M = \{l(\vec{t}\ L), (P_1T_1\ P_2T_2)\}$$

where:

- l is a compound task that represents the loop;
- \vec{t} represents the arguments passed to the loop method;

L is a list representing the collection C ;

P_1 is a first satisfier precondition that contains as their first two queries $head(V\ L)$ and $rest(L'\ L)$, which are used to compute the first item of L and bind it to V , and to compute the remaining items of L and to bind it to L' ;

T_1 represents the sequence of steps S , where any use of I should be made with V , and which contains a task invocation $l(\vec{t}\ L')$ as its last task;

P_2 is trivially true and

T_2 is the empty list,

thus:

$$M = \{l(\vec{t}\ L), (first(head(V\ L)\ rest(L'\ L)) (S\ l(\vec{t}\ L'))\ true\ ()))\}$$

for-each loops with implicit collections

Instead of using an explicit collection in a foreach loop, the logical inference support in the JSHOP2 HTN planner also allows us to use an implicit collection. We can build a “for-each” loop that iterates over a set of terms satisfying a certain condition, without explicitly collecting these terms in a list. In order to represent these loops in JSHOP2 HTNs, we use a method whose precondition comprises the loop’s condition. Persistent variables are used to keep track of the valid substitutions for the condition that has been already visited.

More formally, for a loop of the form:

foreach \vec{V} *satisfying* C *loop* S

where:

\vec{V} is a list of variables from C , serving to forall-quantify these variables;

C is a condition in which the variables of \vec{V} appear;

S represents a sequence of non-behaviour-preserving transformation steps,

the loop is represented with a method M of the form:

$$M = \{l(\vec{t}), (P_1 T_1\ P_2 T_2)\}$$

where:

l is a compound task that represents the loop;

\vec{t} represents the arguments passed to the loop method;

P_1 is a first satisfier precondition that represents the condition C ;

P_1 contains, as its last query, an expression $not(pv(v_{fc}\ \vec{V}))$;

T_1 represents the sequence of steps S and contains, as its last task, a task invocation sequence $(cpv(v_{fc}\ \vec{V})\ l(\vec{t}))$;

P_2 is trivially true and

T_2 contains task invocations for the proper auxiliary operators that will remove the v_{fc} persistent variables: $dav(v_{fc})$,

thus:

$$M = \{l(\vec{t}), (first(C\ not(pv(v_{fc}\ \vec{V}))) (S\ cpv(v_{fc}\ \vec{V})\ l(\vec{t}))\ true\ (dav(V_{fc}))))\}$$

where:

v_{fc} represents a unique persistent variable that references this and only this loop;

v_{fc} serves as a “foreach” control;

$pv(v_{fc} \vec{V})$ is a persistent variable predicate used to “store” the visited \vec{V} values as a list of values, once they have been bound;
 $cpv(v_{fc} \vec{V})$ is a method for committing the visited set of values \vec{V} to the persistent variable v_{fc} and
 $dav(v_{fc})$ is a method for removing all the stored \vec{V} sets for the “for-each” control persistent variable v_{fc} .

Notice that, as in the previous types of loops, the decomposition branch P_2T_2 is intended for exiting the loop.

Apply a non-behaviour-preserving transformation

A transformation step that intends to apply a non-behaviour-preserving transformation is a task invocation $nbpt$ within a task list, and has the form:

$$nbpt(\vec{t})$$

where:

$nbpt$ represents the invocation of the transformation step, which can be either a method or an operator and
 \vec{t} represents the transformation arguments.

As defined in Section 4.3.2, the “apply transformation” invocation that can be used in NBPTs is restricted to invoking other NBPTs.

5.4.4 Non-behaviour-preserving transformations

A non-behaviour-preserving transformation (NBPT) is a complex transformation that can be composed of many steps, including simple steps such as atomic system changes and the invocation of other NBPTs, or compound steps such as loops and alternatives.

In order to represent NBPTs with JSHOP2 HTNs, operators do not suffice. The effect lists of an operator cannot be used to represent transformation algorithms. On the contrary, methods can be further decomposed into other methods or operators. Moreover, their task list decompositions can hold ordered sequences of task invocations. Operators cannot be further decomposed and the order of the predicates in their effect lists is not relevant. As a consequence, any complex transformation should be represented with methods rather than operators.

An NBPT will be represented with a method whose task list decomposition compiles the sequence of steps of the transformation’s algorithm. As defined in Section 4.3.2, an NBPT is not guarded by a precondition and its application may be attempted without any previous checking of the system’s state. Therefore, no condition can be imposed on the corresponding HTN method, so the method’s precondition should be trivially true.

More formally, a non-behaviour-preserving transformation, performed with a sequence of steps S , is a method M of the form:

$$M = \{nbpt(\vec{t}), (P \ T)\}$$

where:

$nbpt$ is a compound task that represents the transformation;
 \vec{t} represents the transformation’s list of parameters;
 P is trivially true, since the transformation has no precondition and

T is a task list representing the sequence of transformation steps S , so it can contain *nbpt*, *del*, *add*, *rep*, *l* or *alt* task invocations.

thus:

$$M = \{nbpt(\vec{t}), (true\ S)\}$$

5.4.5 Refactorings

Refactorings, in the terms described in the previous chapter (see Section 4.3.2), are composed of a precondition, a sequence of transformation steps and can have a parameter list. Given our refactoring's definition, it is quite straightforward to represent refactorings with JSHOP2 HTNs. The representation of refactorings is quite similar to that of NBPTs. The only difference between them is the precondition used in a refactoring to guarantee behaviour preservation.

A refactoring that is performed with a sequence of transformation steps S is specified in JSHOP2 HTNs as a method M of the form:

$$M = \{r(\vec{t}), (P\ T)\}$$

where:

- r is a compound task that represents the refactoring;
- \vec{t} is a list of variables that represent the refactoring's list of parameters;
- P is used to specify the refactoring's precondition and
- T is a task list representing the sequence of transformation steps S , which comprise the refactoring's algorithm.

It should be noticed that, in our model (see Section 4.3.2), a refactoring algorithm can only include NBPT transformation steps, such as applying NBPT transformations *nbpt*-, basic system changes *del*, *add*, *rep*- or composed changes *l*, *alt*.

5.4.6 Invocations and queries of refactoring strategies

Once we have described how the elements of refactorings and non-behaviour-preserving transformations can be written in JSHOP2 HTNs, the next sections are focused on the elements related to refactoring strategies, such as non-deterministic control constructs, etc.

Apply a transformation

Applying a transformation means to invoke a task within a task decomposition list. This construct is similar to "apply an NBPT". It can be represented in exactly the same way as the former, but in this case, any transformation –refactoring strategies, refactorings and NBPTs– can be invoked, as defined in 4.3.3. Formally, a strategy step that intends to apply a transformation is a task invocation h within a task list, and has the form:

$$h(\vec{t})$$

where:

- h represents the invocation of the transformation and
- \vec{t} represents the transformation parameters.
- h can be any transformation kind we have defined: *nbpt*, r or rs (rs is defined in Section 5.4.9 in page 118).

Try a transformation

Trying a transformation means that an attempt should be made to plan a task, but if this does not succeed, a valid plan, which does not include this task, could still be found.

For each transformation $h(\vec{t})$ in the domain, which should be allowed to be invoked with “try”, we have to define two methods M_1 and M_2 :

$$\begin{aligned} M_1 &= \{try(\vec{t}'), (P_1 T_1)\} \\ M_2 &= \{try(\vec{t}'), (P_2 T_2)\} \end{aligned}$$

where:

- try represents the method to try the task h ;
- \vec{t}' is the same list of arguments \vec{t} , but contains the task symbol of the original transformation $-h-$ as its first parameter;
- P_1 is trivially true;
- T_1 contains as its single task $h(\vec{t})$;
- P_2 is trivially true and
- T_2 is the empty list,

thus:

$$M_1 = \{try(h \vec{t}), (true(h(\vec{t})))\} \text{ and } M_2 = \{try(h \vec{t}), (true())\}$$

In order to invoke the transformation $h(\vec{t})$ through “try”, it should be done as:

$$try(h \vec{t})$$

User queries

User queries are implemented as external procedures. A user query is specified with a call term of the form:

$$Q = c((\vec{m} \vec{v})) : T,$$

where:

- c represents the call term;
- \vec{m} represents a list of string literals that will be displayed to the user as a question or request;
- \vec{v} represents a list of input arguments that will be shown to the user to complement the request, and which should be bound when invoking the user query and
- T , as the result of the term evaluation, represents the response fetched from the user.

Duplicated code and entity-kind queries

These are complex queries that are not being supported in this dissertation. They are sufficiently complex for us to consider that they cannot be written solely with HTN axioms, but with a combination of axioms and external procedures.

5.4.7 Non-deterministic alternatives

Non-deterministic alternatives offer multiple selection branches, one of which the planner has to select and apply. The conditions that guard the alternative branches are optional; therefore, if the planner selects a branch with an empty precondition, the success of the branch depends on

the success of the transformation steps in it. The planner will select and attempt to apply all branches non-deterministically until one of them succeeds or else all of them fail.

In order to represent non-deterministic alternatives with JSHOP2 HTNs, we use one method per branch. Each alternative branch is represented as a method whose precondition matches the branch's condition or will be trivially true if the branch does not have a condition. All these branch methods share the same identity –same identifier and same number of parameters–, which will be used to invoke the alternative. Since there is one single task to invoke all the branches, the planner can select one of them non-deterministically when the alternative is invoked.

More formally, a non-deterministic alternative of the form:

$$\text{alt } C_1 S_1 \ C_1 S_1 \ \dots \ C_n S_n$$

where:

each C_i is a branch condition and

S_i is a sequence of transformation steps that can be attempted to instantiate when C_i holds,

is represented with a set of methods M_1, M_2, \dots, M_n , of the form:

$$M_i = \{\text{ndalt}(\vec{t}), (P_i \ T_i)\}$$

where:

each M_i represents the i branch of the non-deterministic alternative;

ndalt represents the alternative;

\vec{t} is a list of arguments comprising the union of all the parameters that have to be passed to all branches;

the method's header $\text{ndalt}(\vec{t})$ is the same for all branch methods, so the alternative is invoked through it;

P_i is a precondition that represents the condition C_i if it is non-empty, or a trivially-true precondition otherwise, and

T_i is a task invocation list that represents the transformation steps from S_i .

5.4.8 Non-deterministic loops

We have defined non-deterministic loops as loops whose success may depend on how their iterations are scheduled. In order to support this, we should let the planner find and select the valid iteration orderings. In JSHOP2 HTNs, this can be easily done. Non-deterministic loops are specified exactly as regular loops, except they do not use the special “first precondition”. This implies that if an iteration ordering does not succeed, the planner will try another valid substitution for the iteration. This will result in the planner attempting different iteration orderings until one succeeds or until all of them fail. As a caveat, it should be noticed that planning for non-deterministic loops might be very inefficient. Exploring every possible iteration ordering for n iterations has an $n!$ branch factor, and thus can greatly widen the search space.

while loops

For a non-deterministic loop of the form

$$\text{while } C \ \text{nd_loop } S$$

the loop is represented with a method M of the form:

$$M = \{ndl(\vec{t}), (P_1 T_1 \ P_2 T_2)\}$$

where:

ndl is a compound task that represents the loop;

\vec{t} represents the argument list passed to the loop method;

P_1 is a precondition that represents the condition C ;

T_1 represents the sequence of steps S and contains a task invocation $ndl(\vec{t})$ as its last task;

P_2 is trivially true and

T_2 is the empty list,

thus:

$$M = \{ndl(\vec{t}), (C (S \ ndl(\vec{t})) \ true \ ())\}$$

With this loop specification, $P_1 \ T_1$ represents the loop's iteration and $P_2 \ T_2$ represents the exit branch.

do-while loops

For a non-deterministic loop of the form:

$$nd_loop \ S \ while \ C$$

the loop is represented with two methods M and M' of the form:

$$M = \{ndl(\vec{t}), (P \ T)\}$$

$$M' = \{ndl'(\vec{t}), (P_1 T_1 \ P_2 T_2)\}$$

where:

ndl is a compound task that represents the loop and that should be used for invoking the loop;

\vec{t} represents the list of parameters passed to the loop method;

P is a precondition that contains all the queries Q needed by the loop's first iteration, and that should be evaluated to true and be independent from the loop's exit condition C ;

T represents the sequence of steps S and contains a task invocation $ndl'(\vec{t})$ as its last task;

ndl' is a compound task that represents the loop's condition and next iteration, and that should be invoked just from ndl ;

P_1 is a precondition that represents the condition C ;

T_1 represents the sequence of steps S and contains a task invocation $ndl'(\vec{t})$ as its last task;

P_2 is trivially true and

T_2 is the empty list,

thus:

$$M = \{ndl(\vec{t}), (Q (S \ ndl'(\vec{t})))\}$$

$$M' = \{ndl'(\vec{t}), (QC (S \ ndl'(\vec{t})) \ true \ ())\}$$

for-each loops with explicit collections

For a non-deterministic loop of the form:

$$foreach \ I \ in \ C \ nd_loop \ S$$

the loop is represented with a method M of the form:

$$M = \{ndl(\vec{t} L), (P_1 T_1 P_2 T_2)\}$$

where:

ndl is a compound task that represents the loop;

\vec{t} represents the arguments passed to the loop method;

L is a list representing the collection C ;

P_1 is a precondition that contains as their first two queries $head(V L)$ and $rest(L' L)$, which are used to compute the first item of L and bound it to V , and to compute the remaining items of L and to bound it to L' ;

T_1 represents the sequence of steps S , where any use of I should be done through V , and which contains a task invocation $ndl(\vec{v} L')$ as its last task;

P_2 is trivially true and

T_2 is the empty list,

thus:

$$M = \{ndl(\vec{t} L), ((head(V L) rest(L' L)) (S ndl(\vec{v} L')) true ()))\}$$

for-each loops with implicit collections

For a non-deterministic loop of the form:

$$foreach \vec{V} \text{ satisfying } C \text{ nd_loop } S$$

the loop is represented with a method M of the form:

$$M = \{ndl(\vec{t}), (P_1 T_1 P_2 T_2)\}$$

where:

ndl is a compound task that represents the loop;

\vec{t} represents the arguments passed to the loop method;

P_1 is a precondition that represents the condition C and contains as its last query an expression $not(pv(V_{fc} (\vec{V})))$;

T_1 represents the sequence of steps S and contains, as its last task, a task invocation sequence $cpv(V_{fc} (\vec{V})) ndl(\vec{t})$;

P_2 is trivially true and

T_2 contains task invocations for the proper auxiliary operators that will remove the V_{fc} persistent variables: $dav(V_{fc})$,

thus:

$$M = \{ndl(\vec{t}), ((C not(pv(v_{fc} \vec{V}))) (S cpv(v_{fc} \vec{V}) ndl(\vec{t})) true (dav(V_{fc}))))\}$$

5.4.9 Unordered strategy steps

Unordered steps are a sequence of transformations whose steps can be applied in an unspecified order. We represent them with JSHOP2 HTNs as a set of separate methods with the same name and number of arguments, each one encapsulating one of the transformation steps. Each method uses persistent variables as “flags” to avoid being re-invoked. The sequence of steps is written as a sequence of as many identical invocations as the number of transformation steps. A last additional step is included in the original intended sequence in order to remove all the persistent variables used.

More formally, an unordered sequence of steps S of the form:

$$S = \text{unordered}(s_1 \ s_2 \ \dots \ s_n)$$

is represented as a sequence of task invocations T and a set of methods M_1, M_2, \dots, M_n , of the form:

$$T = (ut(\vec{a}) \ ut(\vec{a}) \ \dots \ ut(\vec{a})) \ dapv(V) \text{ and} \\ M_i = \{ut(\vec{a}), (P_i \ T_i)\}$$

where:

T is a sequence of n identical invocations of $ut(\vec{a})$, ended by a task $dapv(V)$ for removing all the persistent variables V used for controlling the invocations;
 each M_i encapsulates the transformation step s_i ;
 ut represents an unordered step;
 the methods' header $ut(\vec{a})$ is the same for all the unordered steps, so each one is invoked through it;
 \vec{a} is a list of arguments comprising the union of all the parameters needed by all steps s_i ;
 P_i contains a single predicate $not(pv(V \ i))$ and
 T_i is a task invocation list containing the task t_i that represents the transformation s_i , and a task $cpv(V \ i)$,

thus:

$$M_i = (ut(\vec{a}) \ (not(pv(V \ i))) \ (t_i \ cpv(V \ i)));$$

where:

V is a persistent variable that represents the unordered task list control, and stores which unordered steps have been applied already;
 $pv(V \ i)$ is a persistent variable predicate that states the task t_i has already been applied by the planner;
 $cpv(V \ i)$ is the invocation of a method that stores the persistent variable $pv(V \ i)$ and
 $dapv(V)$ is the invocation of a method for removing all the persistent variables $pv(V \ i)$ used.

Refactoring strategies

Refactoring strategies (see Section 4.3.3) are composed of a precondition, a sequence of strategy steps and can have a parameter list. They are also addressed through achieving a certain goal. In a similar way to refactorings, representing refactoring strategies with JSHOP2 HTNs is straightforward.

More formally, a refactoring strategy rs that addresses a goal g can be specified with a method M of the form:

$$M = \{rs(g \ \vec{t}), (P \ T)\}$$

where:

rs is a compound task that represents a refactoring strategy addressing the goal g ;
 \vec{t} is a list of variables that represents the strategy's list of parameters;
 P is used to specify the strategy's precondition and
 T is a task list representing the sequence of steps which comprise this particular strategy.

As opposed to refactoring's or NBPT's algorithms, the steps in T can refer to any kind of transformation: *nbpt*, *r*, *rs*, *try*, *l*, *alt*, *ndl*, *ndalt*, *add*, *del* or *rep*.

5.4.10 Refactoring strategies specification language as JSHOP2 HTNs

In addition to the previous specifications on how to write the refactoring strategies elements as HTNs, we have formulated a set of systematic rules to translate the refactoring strategies specification language as JSHOP2 HTNs. We have considered that, in order to be useful, the translation of the refactoring strategies specification language has to be described at the language level, using the JSHOP2 syntax directly. In order to improve the readability of the present chapter, these rules have been placed in Appendix B.

5.5 Computing a refactoring planning problem

Having described how refactoring strategies can be written as JSHOP2 HTNs, this section describes how a refactoring strategy is computed and instantiated with JSHOP2.

Planning domain

Refactoring strategies are used to compile, through “recipes”, the available knowledge about how to remove a design smell or to apply a complex refactoring process. These “recipes” can be written and organised with the concepts defined in chapter 4: refactoring strategies, refactorings and non-behaviour preserving transformations.

In order to instantiate a refactoring strategy with an HTN planner, the strategy, and all the associated knowledge needed, has to be specified in an HTN domain. All the operators, methods, axioms and external procedures that comprise the set of available refactoring strategies constitutes the refactoring planning domain.

The planning domain D for the refactoring planning problem is a set of the form:

$$D = \{O, M, A, P\}$$

where:

O is a collection of operators containing at least the following:

- operators implementing the three basic system changes: *add*, *del* and *rep*,
- operators implementing auxiliary tasks for managing persistent variables: *cpv*, *dpv* and *dapv*;

M is a collection of methods representing refactoring strategies *rs*-, refactorings *r*-, non-behaviour preserving transformations *nbpt*-, deterministic and non-deterministic control constructs *l*, *alt*, *try*, *ndl* and *ndalt*- and blocks of unordered steps *ut*-;

A is a collection of axioms representing a variety of system queries, containing at least the following:

- axioms to query persistent variables safely: *qpv*,
- axioms implementing auxiliary functions for managing lists, such as: *head* and *rest*;

P is a collection of external procedures representing a variety of system queries that cannot be implemented as axioms, such as *lexical queries* and *user queries*.

System's initial state

The software system to be transformed has to be fed to the planner as its initial state. As we have mentioned, we use a first-order-logic predicate representation of the system's AST as the system representation for the planner. In order to find a refactoring plan for a given objective and a given software system, we use the current version of the system's AST as the planner's initial state.

More formally, the system's initial state S is a set of the form:

$$S = \{E_1, E_2, \dots, E_n\}$$

where E_i are logical atoms representing the elements of the whole system's AST. All E_i should be ground.

Refactoring planning problem

Finding a refactoring plan to achieving a certain objective, such as removing a bad smell from a software system, is performed through the instantiation of a refactoring strategy that specifies how to address that objective by transforming the system with a behaviour-preserving transformation sequence. The instantiation of a refactoring strategy for a particular case is carried out, in our approach, as the computation of a planning problem formulated over the refactoring planning domain.

More formally, the instantiation of a refactoring strategy rs that addresses a goal g , over a system S , is the computation of a solution for a refactoring planning problem P that models that particular situation.

A refactoring planning problem P is represented as an HTN planning problem of the form:

$$P = \{D, S, G\}$$

where:

D is the refactoring planning domain;

S is the initial system state and

G is a ground task list of the form: $G = (rs(g \vec{p}))$ with rs representing a refactoring strategy that addresses g and \vec{p} representing the strategy's ground parameter list.

The goal G can be defined for addressing multiple sequential objectives by specifying a list of different strategies:

$$G = (rs(g_1 \vec{p}_1) \ rs(g_2 \vec{p}_2) \ \dots \ rs(g_n \vec{p}_n))$$

Refactoring plan

A refactoring plan is a solution to a refactoring planning problem. The refactoring plan is produced by the planner at the lowest level of detail. The plan is given in terms of a sequence of operators, which, in our approach, are atomic changes to the AST. Nevertheless, the more coarse-grained representation of the plan, specified in terms of refactorings, non-behaviour-preserving

transformations and refactoring strategies, is extracted from the decomposition tree that is traversed during the planning process.

A refactoring plan rp is an instantiation of a refactoring strategy rs , if it is a solution for a refactoring planning problem $P = \{D, S, G\}$, where $G = (rs(g \vec{p}))$.

A refactoring plan rp is a sequence of source code transformations. It is given by the planner as a sequence of atomic AST transformations:

$$rp = (sc_1 \ sc_2 \ \dots \ sc_n)$$

where sc_i are atomic ground system changes, either *add*, *del* or *rep* system transformations.

A refactoring plan is more useful when expressed as a sequence of higher level transformations, mainly refactorings. The plan rp can be split into non-overlapping sequences $(sc_j \dots sc_k)$, which are partial plans for G and plans for a higher level transformation t_i . The trace of the decomposition tree traversed by the planner to obtain $(sc_1 \ sc_2 \ \dots \ sc_n)$ as a plan for $rs(g \vec{p})$ can be used to extract this information and represent rp in terms of higher-level transformations.

A refactoring plan rp can be represented as a sequence:

$$rp = (t_1 \ t_2 \ \dots \ t_n)$$

where t_i are system transformations, such as refactorings and non-behaviour-preserving transformations, for which sequences of atomic system changes $(sc_j \dots sc_k)$ are plans for each t_i .

5.6 How JSHOP2 meets the requirements of refactoring planning

In this section, we discuss how HTN forward planning, and specifically the JSHOP2 planner, meets the requirements we have formulated for the refactoring planning problem.

5.6.1 General requirements

Computation.

The relations between refactorings –dependencies and conflicts– are not actually computed by the planner. Instead of that, our approach implements the alternative computational method mentioned in Section 4.5. The dependencies between refactorings are explicitly or implicitly embedded into the refactoring planning domain HTN. The refactoring strategy addressing a certain refactoring would include its dependencies as optional preparatory refactorings. The planner allows us to define full specifications of refactorings that can be executed over the system's state. The plan is computed and generated in the same order as it would be applied. These two features allow us to compute whether those preparatory refactorings would be needed or not. The availability of the current system's state also allows refactoring conflicts to be managed. These are not explicitly included in the HTN, but if a conflicting sequence of transformations is attempted, the planner would detect it by means of unmet preconditions and would therefore select another conflict-free sequence.

Software model.

The AST representation of the software system can be represented as a set of ground logic predicates. JSHOP2 uses a set of ground logic predicates to represent the system's state. Therefore a set of ground logic predicates that represent the software system's AST can be trivially used in the planner to represent the current state of the system.

Deterministic and non-deterministic control constructs.

In this chapter, we have described how control constructs can be expressed as JSHOP2 tasks and thus how they can be used in the planner. The computational model of the JSHOP2 planner, planning in the same order as the tasks should be applied over the system, implies that planning for these tasks, which represent control constructs, is exactly the same as executing the algorithms they implement. JSHOP2 HTNs can be seen as a true programming language and, as we have described in this chapter, these control constructs can be written with it.

Incomplete specifications.

The presented approach enforces the incremental development and formalization of refactoring heuristics. The refactoring planning domain, written as JSHOP2 HTNs, is meant to be incrementally improved. However, the quality of the refactoring plans produced for a given objective depends on the quality of the refactoring planning domain used. Results may vary from finding no plan at all, to obtaining a set of good applicable plans. A more complete refactoring planning domain will lead to better plans for more objectives in more possible situations. A more optimized planning domain will increase the efficiency of the planning process.

Elements of refactoring strategies.

All the elements we have identified to define refactoring strategies can be specified as JSHOP2 task networks to implement the computation of a complex refactoring process as a planning problem. In this chapter, we have demonstrated this by describing how each element can be specified with JSHOP2 HTNs. As an exception to this, we have not addressed duplicate code queries and entity kind queries. The objective of this PhD dissertation is not to develop refactoring strategies themselves, but to propose how they can be supported. Given the complexity of their implementation, we have considered they are beyond the scope of this Thesis. These query types will allow for more advanced refactoring strategies, but they are not, however, needed for demonstrating our approach.

5.6.2 Requirements as a planning problem**Planning assumption A0 (finite number of states)**

The task network helps to prune most of the state space. Even in a scenario of infinite states, this means that the search space can be finite. The computation of a plan in an HTN approach is performed by reducing tasks down the task network. A task network contains a finite number of levels. Each non-primitive task can only be decomposed into a finite number of tasks. Therefore, an acyclic task network defines a finite search space. Recursive or cyclic tasks, which we have used to define loops, can make the state-space infinite, so special caution should be taken when defining them in order to avoid this problem. The precondition of loop methods has to be carefully

written, because a bad precondition can cause “infinite loops” to appear during the planning process. This is always important for every programming language but, in our experience, this is even more important in this case, because HTN domains are hard to debug.

Planning assumption A4 (restricted goals)

HTN planning allows the use of unrestricted goals. The necessary constraints to specify unrestricted goals can be included as domain knowledge specified with HTNs. In our case, one of the constraints for the plans produced by a refactoring planner states that the sequence of transformations represented by the plan should be behaviour-preserving. Refactoring strategies are behaviour-preserving by definition. The search for a plan is performed by strictly following the control rules specified by them. Correctly written refactoring strategies force the planner to produce plans which adhere to the behaviour-preservation constraint.

Chapter 6

Case Study

In order to demonstrate the feasibility of our approach, we have developed a small HTN domain for the refactoring planning problem, which addresses *Feature Envy* and *Data Class* design smells. We have assembled a prototype that uses tools from other researchers and carried out a case study over a set of open-source systems to characterise it. This chapter comprises a description of the refactoring planning domain we have elaborated, the reference prototype we have integrated, and an analysis and discussion of the results obtained with them for the two types of design smells addressed. The design smell correction proposal, based on refactoring planning, that is presented in this Thesis is characterised with the taxonomy defined in Chapter 3 as well.

6.1 A refactoring planning domain

Our main contribution in this prototype is the refactoring planning domain we have written to demonstrate our approach. In order to feed the planner, the specifications of a set of refactorings, refactoring strategies, other transformations and system queries have been represented as JSHOP2 task networks. In practice, this actually means programming these transformations and computations in the JSHOP2 domain definition language. We have used a diagramming tool and a simple text editor to write this domain knowledge. The diagramming tool has been useful for designing the task networks that have been coded with the text editor. Writing and debugging the refactoring planning domain has been the most difficult activity of our approach. However, these specifications can be easily reused. So far, refactoring strategies have been written for: *Remove Data Class*, *Remove Feature Envy* and **MOVE METHOD**; and refactoring specifications for 9 refactorings: **ENCAPSULATE FIELD**, **MOVE METHOD**, **RENAME METHOD**, **RENAME FIELD**, **RENAME PARAMETER**, **RENAME LOCAL VARIABLE**, **REMOVE FIELD**, **REMOVE METHOD** and **REMOVE CLASS**. We have also used the ECLIPSE + JTRANSFORMER combo to develop and debug the main set of system queries, over 150, that we store in a PROLOG file. These system queries and the rest of available elements in the refactoring planning domain we have elaborated are listed in Appendix C. A review of one of the refactorings we have implemented and the three strategies we have developed follows.

6.1.1 Refactorings

As mentioned before, writing refactoring specifications for the planner actually means programming them with the JSHOP2 language. In order to illustrate how refactorings are im-

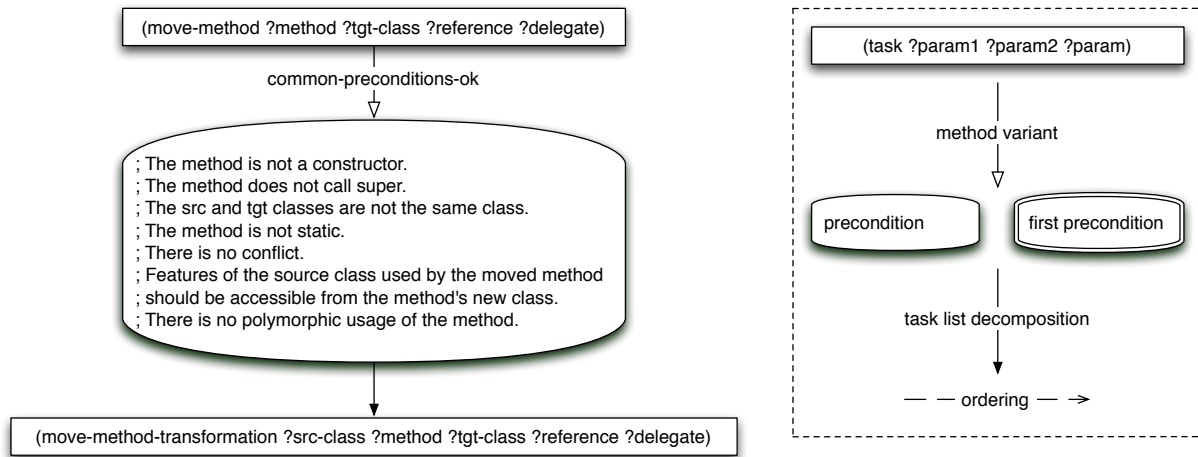


Figure 6.1: Root of the **MOVE METHOD** refactoring design. The top method’s precondition enumerates the conditions common to all the three refactoring’s variations. The right-hand side of the Figure depicts the meaning of the graphical notation used in this and the subsequent **MOVE METHOD** design Figures.

plemented in the refactoring planning domain, we will describe here the **MOVE METHOD** refactoring [FBB⁺99, page 142]. This refactoring’s procedure is briefly described by Fowler *et al.* as: “Create a new method with a similar body in the class it uses most. Either turn the old method into a single delegation, or remove it altogether.”

The refactoring’s mechanics will be reviewed in more detail as we describe its implementation for the refactoring planning domain. This description mimics the actual process we have followed. We have used a diagram editor to help us design the HTNs. Then we have implemented each HTN branch and method with a text editor. We have performed separate tests for each method and comprehensive tests for the full refactoring. The implementation of the **MOVE METHOD** spans over 30 JSHOP2 methods.

In order to implement the refactoring, we have split its mechanics into three separate HTN branches: move the method and reference it through a field of the source class, move the method and reference it through a parameter of the original method, and move the method and keep a delegating method in the original class. Some initial preconditions have to be checked before trying to apply the refactoring. We have factored out the preconditions common to the three different variations of the refactoring and placed them in the top entry point of the refactoring. The diagram of this design is depicted in Figure 6.1. Listing 6.1 shows the implementation of the **MOVE METHOD** refactoring root method.

We have defined four arguments for this refactoring: `?method` references the method to be moved; `?tgt-class` references the target class where the method shall be moved to; `?reference` relates to the entity which is going to be used to reference the moved method, it can be either a field of the original class or a parameter of the original method; finally, `?delegate` is a kind of boolean parameter indicating whether a delegating method should be kept in the original class or not. It is expected to be either `true` or `false`. As described in Figure 6.1, the refactoring’s top method establishes that if the method’s preconditions are met, the task `move-method` can be achieved by decomposing it into the subtask `move-method-transformation`, which defines the actual transformation. A new parameter `?src-class` has been added to reference the source

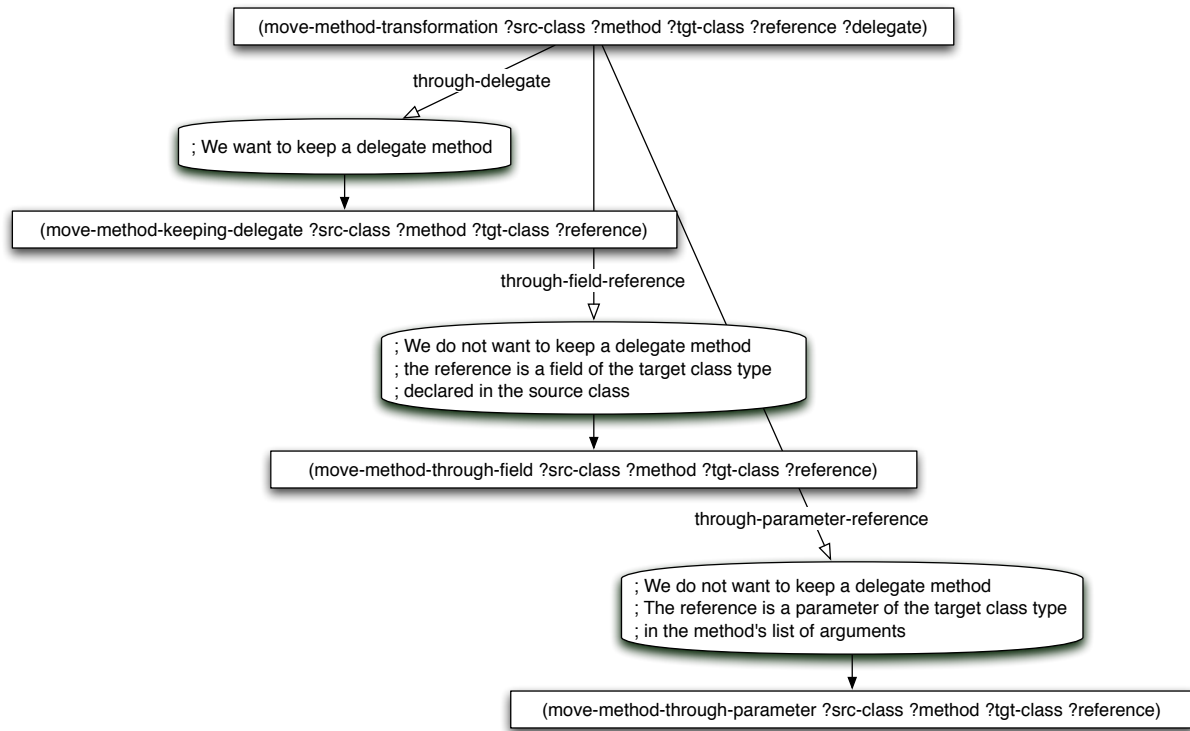


Figure 6.2: Design of the decomposition of the task `move-method-transformation` into the three different variations of the **MOVE METHOD** refactoring.

class of the method to be moved. This is a relevant piece of information needed to perform the transformation. Once this reference to the source class has been computed during the evaluation of the preconditions, this information can be reused. The reference is passed as an argument to avoid re-computing it again. This is a common practice we have used while writing refactorings for the refactoring planning domain.

The three possible variations of the refactoring have been designed as three different alternative methods. This is shown in Figure 6.2. This design states that the method at the top of the figure, `move-method-transformation`, can be achieved by applying one of the three alternative method decompositions: `move-method-keep-delegate`, `move-method-through-field` or `move-method-through-parameter`. No order has been specified, so these alternatives are non-deterministic. Additional preconditions have been defined to guard each method decomposition, therefore, the planner would only attempt to apply those whose precondition holds. The implementation of this non-deterministic alternative method decomposition is displayed in Listing 6.2. In order to define a non-deterministic alternative, we have written three different methods that share the same task symbol `move-method-transformation`. They specify three different method decompositions for achieving this task. Each method contains a different precondition and task decomposition. Each decomposition specifies a different list of subtasks to be applied in order to achieve the `move-method-transformation` task.

Since the full implementation of the **MOVE-METHOD** refactoring is too big to be completely reviewed here, from this point on, we will only traverse and describe in detail, at each task

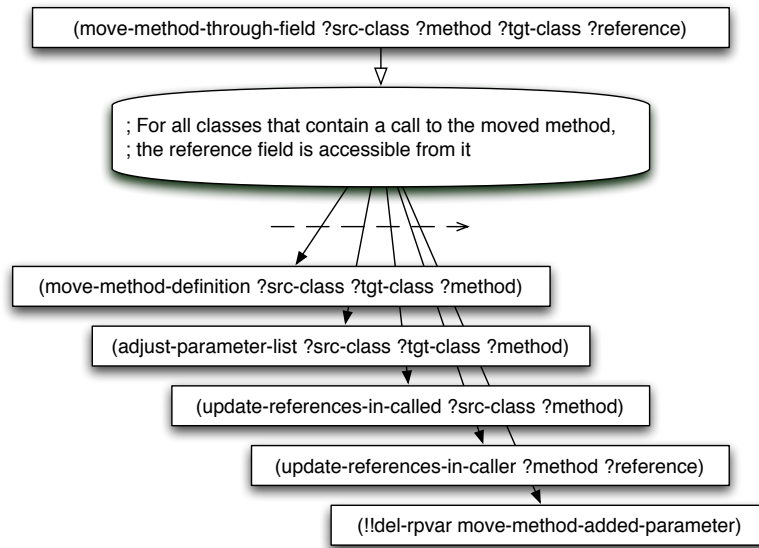


Figure 6.3: Design of the **MOVE METHOD** refactoring variation that moves the method to a target class referenced through a field of the source class. If the precondition holds, this task has to be achieved by applying five subtasks in the specified order.

network level, the task decomposition and just one of its subtasks. As for the three variations of the refactoring, we will focus on `move-method-through-field`, the one that attempts to move a method completely, using a field of the original class as a reference and without keeping a delegate method. Figure 6.3 represents the design of this variation’s implementation. We have defined a single decomposition to achieve the task `move-method-through-field`. This decomposition is guarded by a precondition which is specific to this particular **MOVE-METHOD** variant. The task list of this decomposition is composed of five tasks that have to be applied in the given order. The first four tasks actually implement the transformation, while the last is aimed at deleting a persistent variable `move-method-added-parameter` that is used during the plan computation.

The first four tasks define a basic procedure for moving the method. Firstly, the method definition has to be moved from the source class to the target class. This implies modifying the method’s header and the involved classes, so the AST references between the original class and the method are removed, and the new AST references between the method and the destination class are added. When moving a method from a source class to a target class, a new parameter of the source class type is added to the method. This parameter is used to pass a reference to an object of the source class type to the method so it can still access all the features it uses from this class. The next task defines a transformation for modifying all the references to objects of the original class types within the moved method. These references are changed so they are performed through the newly added parameter, instead of through implicit accesses –unqualified access– or through explicit accesses –using the explicit “this” receiver. Finally, the fourth task will describe how to update all the existing calls to the moved method. These references have to be re-routed through the requested field, in order to point them to the method in its new place within the target class. The implementation of the task network from Figure 6.3 is performed with a single method, which is displayed in Listing 6.3 (page 134).

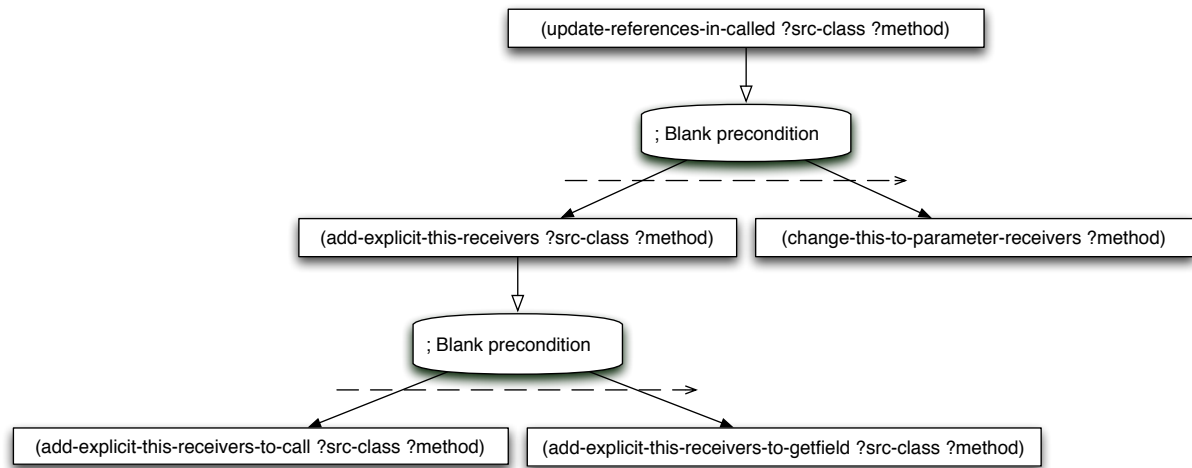


Figure 6.4: Design of a transformation for updating the references to the source class within the moved method during a **MOVE METHOD** refactoring.

The review of the refactoring implementation continues by describing in detail one of the subtasks in the next step down the task network. The `update-references-in-called` task is dedicated to updating the references within the moved method. The design of this task is illustrated in Figure 6.4. All the accesses to the features of the source class within the moved-method should be performed through the added parameter. The `update-references-in-called` task implements this part of the refactoring. It is decomposed into a task list with two subtasks. Firstly, all unqualified accesses are turned into qualified accesses through the explicit “this” receiver. This is performed by the `add-explicit-this-receivers` task. Afterwards, all occurrences of “this” are substituted by accesses to the newly added parameter, which references the source class. This transformation is defined in the `change-this-to-parameter-receivers` task. Finally, the `add-explicit-this-receivers` task is further decomposed into a task list of two more subtasks. One of them implements the addition of “this” receivers to implicit method calls, while the other performs the same operation over implicit field accesses. The task lists in this design should be applied in the specified order and no preconditions are needed to guard any of these decompositions. The implementation of these tasks is shown in Listing 6.4 (page 134). These two levels of the task network, which are fairly easy to implement, are supported by two simple methods, each containing blank preconditions and a single decomposition.

The review of the **MOVE METHOD** refactoring implementation will be completed by describing some of the methods at the lowest level of this refactoring’s task network. These methods are not decomposed into other methods, but into primitive tasks or operators which specify the actual transformations of the system’s state.

The `add-explicit-this-receivers` task, whose design is depicted in Figure 6.5, represents a deterministic while loop. As presented in the former chapter, this kind of loop is designed as a task with two ordered alternative decomposition. One of them specifies the loop’s body and condition and the other serves as the exit branch of the loop. The first decomposition’s branch, which is represented with the `calls-with-implicit-receiver` task, is always evaluated in the first place due to the alternative decompositions being ordered. The exit branch, represented

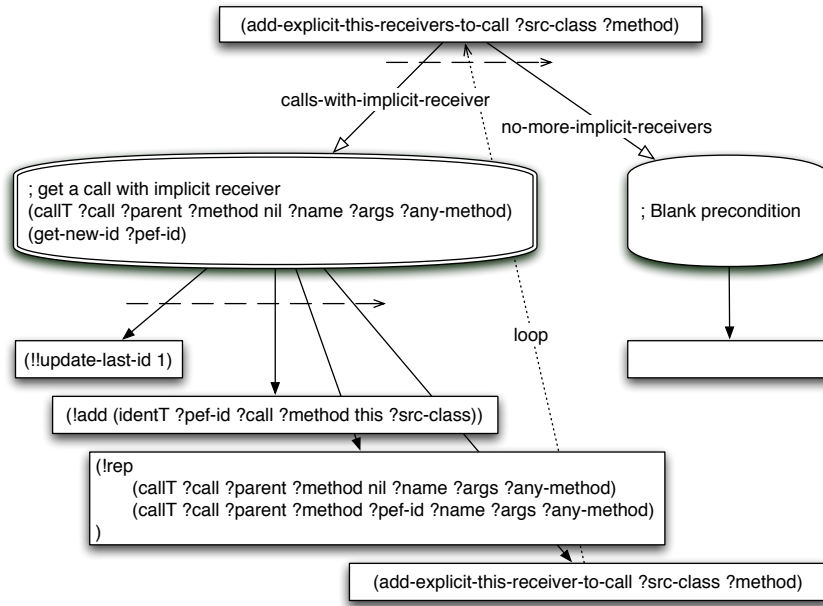


Figure 6.5: Design of a transformation for adding a “this” reference to all unqualified calls to the source class from the moved method during a **MOVE METHOD** refactoring.

with the `no-more-implicit-receivers` task, is only attempted when the first one fails. The implementation of this task is shown in Listing 6.5.

The condition in the first branch looks for an unqualified call within the method, and obtains the full predicate that represents this call. This is a `callT` predicate whose third term –its enclosing method– is the current method, and whose fourth term –its receiver– is the empty list –represented with `nil`. If some unqualified call is found, the next condition –`get-new-id`– is used for computing a new identifier for a new predicate that will be added to the system’s state.

The loop’s body task decomposition is a task list containing four tasks. Firstly, an auxiliary operator is used to increment an internal counter that logs the newly created predicate. The second operator in the task list adds this predicate, which represents the “this” receiver, to the system’s state. The third operator in the task list modifies the current call predicate by linking it to the explicit “this” receiver. The fourth element in the task list is a recursive invocation of the current task that represents entering the next iteration of the loop.

The `add-explicit-this-receiver-to-call` will continue to be invoked recursively until the precondition fails. That is, when no more calls with implicit receivers can be found within the method. The loop’s precondition is marked as a “first” precondition. If an iteration fails, the plan will not try to meet the precondition with a different `call` predicate. This implies that the planner should not attempt different orderings for the loop iterations in order to obtain a valid plan for the loop as a whole. When the loop condition no longer holds, the planner will select the second branch. It will evaluate a blank precondition, which is trivially true, and will apply an empty task list, thus completing the planning process for the `add-explicit-this-receiver-to-call` task.

The `change-this-to-parameter-receivers` is reviewed here in order to give a complete vision of how the accesses from the moved method are finally updated to point to the parameter referencing an object of the original class type. This task implements the transformation of all

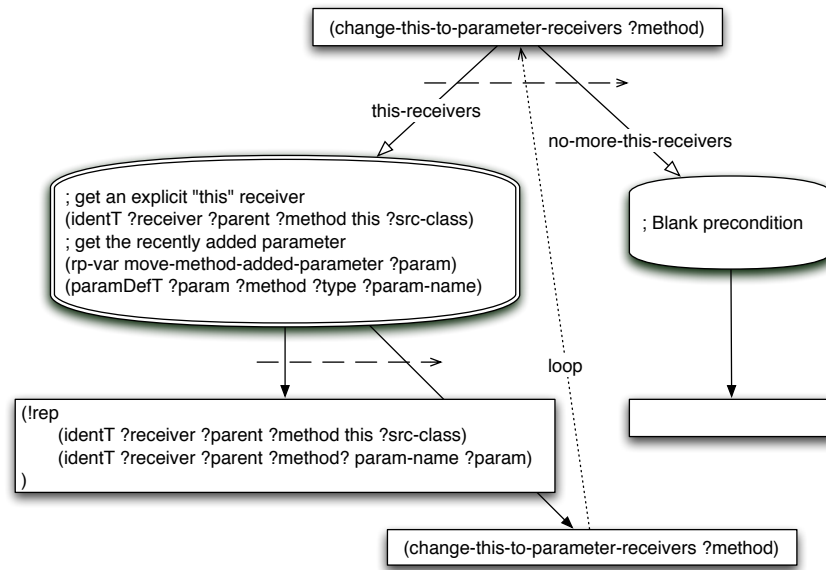


Figure 6.6: Design of a transformation for replacing all “this” references within a moved method to the method’s parameter referencing its source class during a **MOVE METHOD** refactoring.

the “this” accesses into parameter accesses. The design of this task is quite similar to the previous one and is illustrated in Figure 6.6. This task network also represents a while loop.

In this case, the loop’s condition is aimed at gathering all the explicit “this” receivers – `identT` predicates. If the first condition holds, the next condition is used to query a persistent variable, in order to find out which parameter references an object of the original class type. This parameter was added to the method at a previous stage, more specifically, within the `adjust-parameter-list` task (see Figure 6.3). The loop’s body is a task list decomposition of two tasks. The first one is an operator that replaces the “this” receiver, for the parameter. The second task is the recursive invocation of the next iteration of the loop. The implementation of the `change-this-to-parameter-receivers` task can be found in Listing 6.6 (page 136).

6.1.2 Refactoring strategies

We have compiled some basic strategies into HTN domain knowledge: a strategy for applying the **MOVE METHOD** refactoring, and two strategies for removing *Feature Envy* and *Data Class* design smells. They are reviewed in this section. Refactoring strategies can be extracted from the literature and compiled into HTN domain knowledge. The strategies presented here are a subset selected from those that can be found in some catalogues [FBB⁺99, LM06]. They have been translated into the JSHOP2 domain definition language and used to test the suitability of this approach. They are not complete, since the objective of this work is to provide an approach to support refactoring planning, but not to develop the refactoring strategy specifications that can be applied with this technique. As we have already mentioned, our approach allows the incremental improvement of the planner by adding more specifications.

```

1  (:method (move-method ?method ?tgt-class ?reference ?delegate)
2    ( ; Precondition
3      ; The method is not a constructor
4      (not (is_constructor ?method))
5
6      ; The method does not call super
7      (not (call-to-super ?method ?super-method ?super-call))
8
9      ; The src and tgt classes are not the same class
10     (enclosing-class-of-method ?method ?src-class)
11     (not (same ?src-class ?tgt-class))
12
13     ; The method is not static
14     (not (is-static ?method))
15
16     ; There is no conflict
17     (not (is-name-conflict ?tgt-class ?method))
18
19     ; Features of the source class used by the moved method
20     ; should be accessible from the method's new class
21     (forall (?src-class-method)
22       (
23         (method-call-from ?method ?src-class-method ?call)
24         (contains-method ?src-class ?src-class-method)
25       )
26       (can-access-method ?src-class-method ?tgt-class)
27     )
28     (forall (?src-class-field)
29       (
30         (field-access-from ?method ?src-class-field ?access)
31         (contains-field ?src-class ?src-class-field)
32       )
33       (can-access-field ?src-class-field ?tgt-class)
34     )
35   )
36
37   ( ; Task decomposition
38     (move-method-transformation ?src-class ?method ?tgt-class
39                               ?reference ?delegate)
40   )
41 )

```

Listing 6.1: *Root of the MOVE METHOD refactoring implementation. The precondition includes queries to gather the full information of the involved entities, and the preconditions common to the different move method variations.*

```

1  (:method (move-method-transformation ?src-class ?method ?tgt-class
2                                     ?reference ?delegate)
3    through-delegate
4    ( ; Precondition
5      ; We want to keep a delegate method
6      (same ?delegate true)
7    )
8    ( ; Task decomposition
9      (move-method-keeping-delegate ?src-class ?method ?tgt-class ?reference)
10   )
11 )
12
13 (:method (move-method-transformation ?src-class ?method ?tgt-class
14                                     ?reference ?delegate)
15   through-field-reference
16   ( ; Precondition
17     ; We do not want to keep a delegate method
18     (same ?delegate false)
19
20     ; the reference is a field of the target class type,
21     ; declared in the source class
22     (declares-field ?src-class ?reference)
23     (field-type ?reference ?tgt-class)
24   )
25   ( ; Task decomposition
26     (move-method-through-field ?src-class ?method ?tgt-class ?reference)
27   )
28 )
29
30 (:method (move-method-transformation ?src-class ?method ?tgt-class
31                                     ?reference ?delegate)
32   through-parameter-reference
33   ( ; Precondition
34     ; We do not want to keep a delegate method
35     (same ?delegate false)
36
37     ; the reference is a parameter of the target class type,
38     ; in the method's list of arguments
39     (method-parameter ?method ?reference)
40     (parameter-type ?reference ?tgt-class)
41   )
42   ( ; Task decomposition
43     (move-method-through-parameter ?src-class ?method ?tgt-class ?reference)
44   )
45 )

```

Listing 6.2: *Different method decompositions, implemented as separate methods, that implement the three variations for performing a **MOVE METHOD** refactoring.*

```

1 (:method (move-method-through-field ?src-class ?method ?tgt-class ?reference)
2   move-method-through-field-reference
3   ( ; Precondition
4     ; For all classes that contain a call to the moved method,
5     ; the reference field is accesible from it
6     (forall (?caller-class)
7       (
8         (method-call-from ?caller-method ?method ?call)
9         (enclosing-class-of-method ?caller-method ?caller-class)
10        )
11      (can-access-field ?reference ?caller-class)
12    )
13  )
14  ( ; Task decomposition
15    (move-method-definition ?src-class ?tgt-class ?method)
16    (adjust-parameter-list ?src-class ?tgt-class ?method)
17    (update-references-in-called ?src-class ?method)
18    (update-references-in-caller ?method ?reference)
19    (!!del-rpvar move-method-added-parameter)
20  )
21 )

```

Listing 6.3: Method that implements a **MOVE METHOD** refactoring by referencing it in its destination class, through a field of its original class. This variation of the refactoring does not leave a delegate method.

```

1 (:method (update-references-in-called ?src-class ?method)
2   () ; Blank precondition
3
4   ( ; Task decomposition
5     (add-explicit-this-receivers ?src-class ?method)
6     (change-this-to-parameter-receivers ?method)
7   )
8 )
9
10 (:method (add-explicit-this-receivers ?src-class ?method)
11   () ; Blank precondition
12
13   ( ; Task decomposition
14     (add-explicit-this-receivers-to-call ?src-class ?method)
15     (add-explicit-this-receivers-to-getfield ?src-class ?method)
16   )
17 )

```

Listing 6.4: Methods that decompose the task of updating the references within the moved method.

```

1  (:method (add-explicit-this-receivers-to-call ?src-class ?method)
2    calls-with-implicit-receivers
3    (:first ; First satisfier precondition
4      (
5        ; get a call with implicit receiver
6        (callT ?call ?parent ?method nil ?name ?args ?any-method)
7        (get-new-id ?pef-id)
8      )
9    )
10   ( ; Task decomposition
11     (!!update-last-id 1)
12
13     ; add an explicit receiver and replace the call to use it
14     (!add (identT ?pef-id ?call ?method this ?src-class))
15     (!rep
16       (callT ?call ?parent ?method nil ?name ?args ?any-method)
17       (callT ?call ?parent ?method ?pef-id ?name ?args ?any-method)
18     )
19
20     ; loop
21     (add-explicit-this-receiver-to-call ?src-class ?method)
22   )
23
24   no-more-implicit-receivers
25   () ; Blank precondition
26   () ; Empty task decomposition
27 )

```

Listing 6.5: Method that implements a loop for replacing all the implicit unqualified receivers within the moved method by explicit “this” receivers instead.

```

1  (:method (change-this-to-parameter-receivers ?method)
2    this-receivers
3    (:first ; First satisfier precondition
4      (
5        ; get an explicit "this" receiver
6        (identT ?receiver ?parent ?method this ?src-class)
7
8        ; get the recently added parameter
9        (rp-var move-method-added-parameter ?param)
10       (paramDefT ?param ?method ?type ?param-name)
11      )
12    )
13    ( ; Task decomposition
14      (!rep
15        (identT ?receiver ?parent ?method this ?src-class)
16        (identT ?receiver ?parent ?method? param-name ?param)
17      )
18
19      ; loop
20      (change-this-to-parameter-access ?method)
21    )
22
23    no-more-this-receivers
24    () ; Blank precondition
25    () ; Empty task decomposition
26  )

```

Listing 6.6: Method that implements a loop for replacing all the explicit “this” receivers within the moved method by references to the parameter pointing to the method’s original source class.

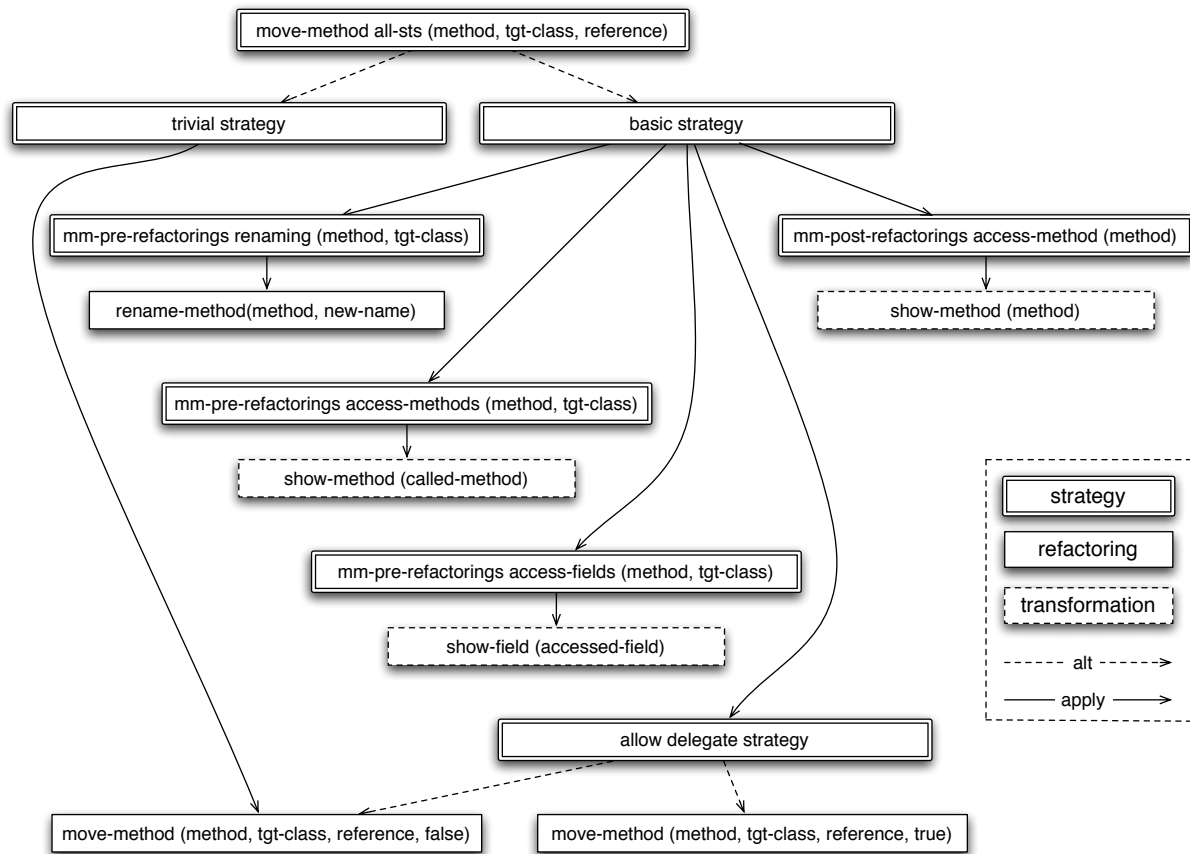


Figure 6.7: Overview of a simple strategy for applying the **MOVE METHOD** refactoring.

A **MOVE METHOD** application strategy

A simple strategy for applying the **MOVE METHOD** refactoring is described here. The design of the strategy is displayed in Figure 6.7 and its specification with our strategy definition language is represented in Listings 6.7 and 6.8.

A **MOVE METHOD** refactoring can be attempted by two alternative strategies: a trivial strategy that would simply try to apply the refactoring as it is, and a basic strategy, that will try to perform some preparatory transformations before trying to apply the refactoring. The trivial strategy will try to apply the refactoring without keeping a delegate method in the source class. The top strategy, which in this case has been labelled “all-sts”, is not meant to impose any ordering or precondition over the alternate strategies it comprises. Therefore, the planner can freely choose from among the available strategies freely. The basic strategy is composed of five sequential substrategy invocations. The first one `-mm-pre-refactorings renaming-` will attempt to solve any name conflicts that might surface when moving the method to the target class. This will be addressed by computing a new name for the method and invoking a **RENAME METHOD** refactoring. The second and third substrategies are aimed at modifying the visibility of any class members, methods or fields that will not otherwise be visible from the moved method at its new location. These lead to transformations that will simply turn these

```

strategy move-method all-sts (method, tgt-class, reference)
  body
    alt
      branch apply move-method trivial (method, tgt-class, reference)
      branch apply move-method basic (method, tgt-class, reference)
    end
  end

strategy move-method trivial (method, tgt-class, reference)
  body
    apply move-method (method, tgt-class, reference, false)
  end

strategy move-method basic (method, tgt-class, reference)
  body
    apply mm-pre-refactorings renaming (method, tgt-class)
    apply mm-pre-refactorings access-methods (method, tgt-class)
    apply mm-pre-refactorings access-fields (method, tgt-class)
    apply move-method allow-delegate (method, tgt-class, reference)
  end

strategy move-method allow-delegate (method, tgt-class, reference)
  body
    alt
      branch apply move-method (method, tgt-class, reference, false)
      branch apply move-method (method, tgt-class, reference, true)
    end
  end

```

Listing 6.7: A strategy defining different approaches that increase the chances of applying the Move Method refactoring successfully.

members to public visibility. After scheduling these preparatory transformations, the fourth substrategy in the basic strategy will try to apply the Move Method refactoring. This will be attempted by two alternative substrategies in no particular order. One of them will try to apply the **MOVE METHOD** refactoring without keeping a delegate method, and the other will invoke the refactoring, explicitly forcing the planner to leave the delegate. Finally, the fifth substrategy in the basic strategy will modify the visibility of the moved-method if needed.

A *Feature Envy* strategy

A method that suffers from *Feature Envy* is a method that accesses and manipulates data from other classes rather than its own. It accesses fields directly or through accessor methods. This smell can be repaired by moving the method to the class whose fields it uses most or by reallocating the accessed data into the method's class. It can also be noticed that there is a relationship between the *Feature Envy* and *Data Class* design smells [LM06]. If a Feature Envy method is accessing a Data Class, it is highly desirable to move the method to that class. A graphical overview of the strategy for removing a *Feature Envy* smell is shown in Figure 6.8.

```

strategy mm-pre-refactorings renaming (method, tgt-class)
  body
    if is-name-conflict (tgt-class, method) then
      alt
        branch
          compute-new-name (method, new-name)
        branch
          method-fqn(method, src-pkg-name, src-class-name, method-name)
          class-fqn(tgt-class, tgt-pkg-name, tgt-class-name)
          user-query("New name for method",
                    (tgt-pkg-name, tgt-class-name, method-name), new-name)
        end
      apply rename-method(method, new-name)
    end
  end

strategy mm-pre-refactorings access-methods (method, tgt-class)
  body
    foreach called-method
      satisfying
        method-call (method, called-method) and
        not(can-access-method (called-method, tgt-class))
      loop
        apply show-method (called-method)
      end
    end
  end

strategy mm-pre-refactorings access-fields (method, tgt-class)
  body
    foreach accessed-field
      satisfying
        field-access (method, accessed-field) and
        not(can-access-field(accessed-field, tgt-class))
      loop
        apply show-field (accessed-field)
      end
    end
  end

strategy mm-post-refactorings access-method (method)
  body
    if method-call(caller-method, method) and
        enclosing-class-of-method(method, caller-class) and
        not(can-access-method(method, caller-class))
    then
      apply show-method(method)
    end
  end

```

Listing 6.8: *Strategies defining the additional transformations that can be applied before a MOVE METHOD refactoring.*

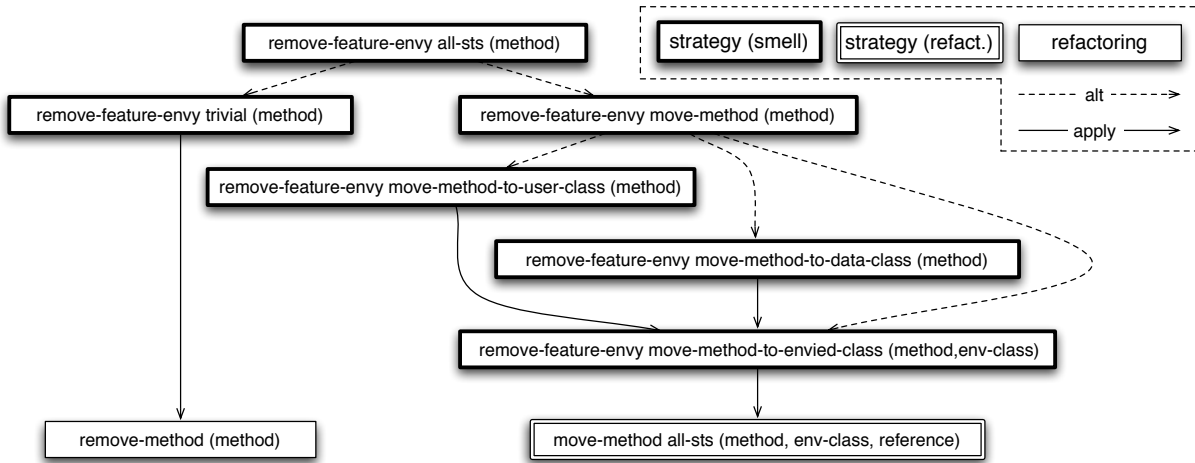


Figure 6.8: Overview of a simple strategy for removing a *Feature Envy* smell.

Figure 6.8 portrays an overview of the relationships between the *Feature Envy* strategies and the refactorings involved. Refactoring strategies are labelled with a goal and a name. Most strategies are labelled with a `remove-feature-envy` goal. One of them `-trivial-` describes a trivial recipe: removing a *Feature Envy* smell by removing the affected method. Another strategy `-move-method-` specifies that the method should be moved to another class. The strategy named `all-sts` is used as the “entry point” one. This is the strategy that should be invoked for instantiating a `remove-feature-envy` strategy. The planner computes the applicability of each alternative strategy and selects the suitable one in order to include it in the refactoring plan.

The difference between applying a refactoring directly instead of through the invocation of an associated refactoring strategy can be noticed in the example. In our example, the **REMOVE METHOD** refactoring is invoked directly, while the **MOVE METHOD** refactoring is invoked through a refactoring strategy. A direct application of the **REMOVE METHOD** refactoring will only succeed when its precondition is met. In this case, we do not want the planner to apply any additional transformation. On the other hand, the application of the **MOVE METHOD** refactoring through one of its refactoring strategies will result in some preparatory transformations being planned before the refactoring is attempted, in order to enable the refactoring’s precondition. In this case, we want the planner to examine all the available ways of performing the Move Method refactoring, even if this implies applying some additional refactorings, because we want this refactoring to be applied by any means.

The alternatives under `remove-feature-envy move-method` represent the strategies that are available for removing the design smell by moving the method to another class. These alternatives are mainly focused on how the target class of the **MOVE METHOD** refactoring can be computed. A closer look at how this is done is displayed in Listing 6.9.

Three different heuristics can be used and each one uses a different query to achieve it. We can simply obtain the candidate target class by querying the system for all the classes accessed from the feature envy method. This heuristic is specified in the substrategy named `remove-feature-envy move-method-to-envied-class`. The `get-envied-class` query is used to obtain those candidate classes. The planning mechanism checks all suitable solutions until one which works is found or until no more possibilities are left.

```

strategy remove-feature-envy all-sts (method)
  body
    alt
      branch apply remove-feature-envy trivial (method)
      branch apply remove-feature-envy move-method (method)
    end
  end

strategy remove-feature-envy trivial (method)
  body
    apply remove-method (method)
  end

strategy remove-feature-envy move-method (method)
  body
    alt
      branch
        apply remove-feature-envy move-method-to-user-class (method)
      branch
        apply remove-feature-envy move-method-to-data-class (method)
      branch
        apply remove-feature-envy move-method-to-envied-class (method, env-class)
    end
  end

strategy remove-feature-envy move-method-to-user-class (method)
  precondition
    method-fqn(method, pkg-name, class-name, method-name)
    user-query("Package name of new class for method",
              (pkg-name, class-name, method-name),
              tgt-pkg-name)
    user-query("Class name of new class for method",
              (pkg-name, class-name, method-name),
              tgt-class-name)
    class-fqn(tgt-class, tgt-pkg-name, tgt-class-name)
  body
    apply remove-feature-envy move-method-to-envied-class (method, tgt-class)
  end

strategy remove-feature-envy move-method-to-data-class (method)
  precondition
    smell("data class", env-class)
    get-envied-class (method, env-class)
  body
    apply remove-feature-envy move-method-to-envied-class (method, env-class)
  end

strategy remove-feature-envy move-method-to-envied-class (method, env-class)
  precondition
    get-envied-class (method, env-class)
    get-movemethod-reference (method, env-class, reference)
  body
    apply move-method all-sts (method, env-class, reference)
  end

```

Listing 6.9: *Draft of some strategies that may be defined to remove a Feature Envy method.*

The `move-method-to-data-class` strategy specifies checking *Data Classes* as candidate target classes. The `smell("data-class", env-class)` query uses the information provided by iPlasma, to find classes labelled as *Data Classes*. Once a *Data Class* is selected, it is passed as an argument to the `remove-feature-envy move-method-to-envied-class` strategy. Since a solution for the variable `env-class` has already been found, in this case, the `get-envied-class` query does not compute the target class but verifies that the selected *Data Class*s can be identified as such.

The remaining strategy `-remove-feature-envy move-method-to-user-class-` employs a user query to compute the candidate target class. A dialog is displayed to the developer requesting the necessary information. The message shown to the user is represented as the first set of arguments, and the information obtained is collected in the second set of arguments.

A *Data Class* strategy

A *Data Class* is a class that contains little functionality and whose data is excessively accessed from other classes. The symptoms of a *Data Class* are that it exposes a lot of public fields; it declares very few methods, or mainly accessor methods, and it presents low complexity. In order to remove a *Data Class* one would try to follow the general approach of bringing behaviour and data close together. If the *Data Class* is trivially simple and it is being accessed from just a single method or class, it would be desirable to move the *Data Class* members to its client class and, if possible, to get rid of the *Data Class*. Moreover, if the class is not being used at all, it could even be completely removed. If the *Data Class* is being used from several clients and the class is worth keeping, one should rather move their client methods to the *Data Class*. The relation between the *Feature Envy* and the *Data Class* smells also plays an important role here. When moving client methods to the *Data Class*, those client methods suffering from *Feature Envy* should preferably be moved.

The strategies we have implemented for removing a *Data Class* are a little more complex than the ones for *Feature Envy*. The overview of the strategy we have defined, within the refactoring planning domain, for removing *Data Class* smells is shown in Figure 6.9.

The top-most strategy, that acts as the entry point for all the available *Data Class* strategies, has been labelled `remove-data-class all-sts`. In order to instantiate this, the planner can choose between two different strategies. The simplest one, `remove-data-class trivial`, attempts to erase the *Data Class* from the system by removing the affected class, thus invoking the **REMOVE CLASS** refactoring. As in the *Feature Envy* case, the **Remove Class** refactoring is invoked directly, because we only want this strategy to be applied when the current system's state meets the refactoring's preconditions. The other alternative *Data Class* strategy `-remove-data-class reorganize class-` comprises a basic class reorganization strategy that is achieved by performing three substrategies.

The first substrategy attempts to move all the *Feature Envy* methods which are clients of the *Data Class*. In order to do that, the `remove-data-class move-fe-methods` deals with finding and gathering all the candidate methods. For each method, it invokes the `remove-feature-envy move-method-to-envied-class`. This is a strategy that was already developed as part of the *Feature Envy* removal strategies, and that is being reused here. The second *Data Class* substrategy `-remove-data-class move-client-methods-` attempts to move the other regular client methods of the data class. These two substrategies invoke the **MOVE METHOD** strategy, so for each candidate method, all available strategies for moving it are checked.

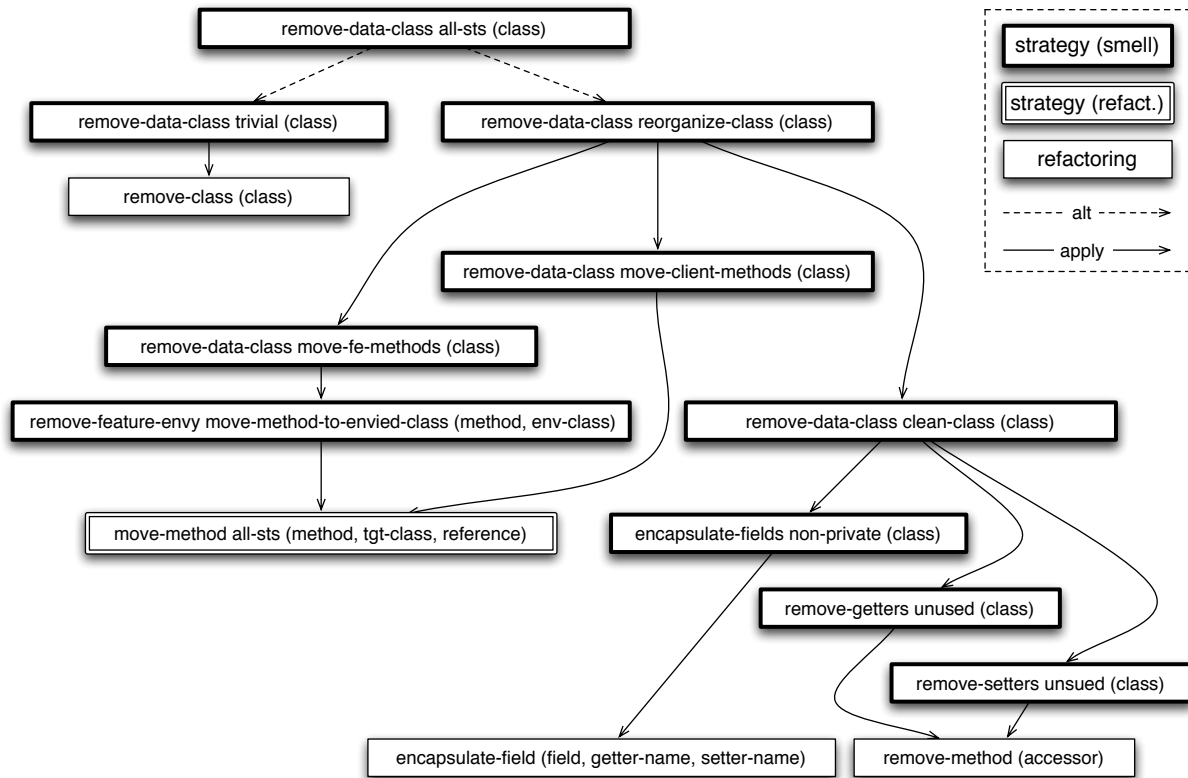


Figure 6.9: Overview of simple strategies for removing a Data Class.

The third substrategy `remove-data-class clean-class` is dedicated to modifying the target class members that may be causing the class to be identified as a *Data Class*. This substrategy decomposes into three steps. Firstly, `encapsulate-fields non-private` is used for gathering all the publicly exposed fields and **ENCAPSULATE FIELD** is invoked to hide all of them. Then, all the unused accessors, which may also be exposing the data class fields too much, are removed. Two strategies `remove-getters unused` and `remove-setters unused` are defined for this purpose. They simply find all the class accessors and invoke the **REMOVE METHOD** refactoring directly over them. Therefore, all those accessor methods which would meet this refactoring precondition, those which are unreferenced and hence unused, are planned to be removed.

The implementation of the strategies we have written for removing a *Data Class* is detailed in Listings 6.10, 6.11 and 6.12. It is worth noticing the usage of the `try` invocation in some circumstances, instead of the `apply` invocation. For example, in Listing 6.11, using the `try` invocation within the loops' bodies implies that the desired strategy will be attempted to be planned for each method satisfying the given conditions. Nevertheless, we do not want the planner to fail if these strategies cannot be instantiated due to one of these methods failing. In order to avoid this issue, the written specification tells the planner to apply the strategy over as many methods as possible. The planner is commanded to “try” these strategies but not to fail if it is not possible to apply them for all cases.

```

strategy remove-data-class all-sts (class)
  body
    alt
      branch
        apply remove-data-class trivial (class)
      branch
        apply remove-data-class reorganize-class (class)
      end
    end

strategy remove-data-class trivial (class)
  body
    apply remove-class (class)
  end

strategy remove-data-class reorganize-class (class)
  body
    apply remove-data-class move-fe-methods-to-class (class)
    apply remove-data-class move-client-methods-to-class (class)
    apply remove-data-class clean-class (class)
  end

```

Listing 6.10: *Top strategies that we have defined to remove a Data Class smell.*

```

strategy remove-data-class move-fe-methods-to-class (class)
  body
    foreach fe-method
      satisfying
        smell("feature envy", fe-method)
        get-envied-class (fe-method, class)
      loop
        try remove-feature-envy move-method-to-envied-class (method, class)
      end
    end
  end

strategy remove-data-class move-client-methods-to-class (class)
  body
    foreach method
      satisfying
        client-method-of-class(method, class)
      loop
        try move-method all-sts (method, class, reference)
      end
    end
  end

```

Listing 6.11: *Strategies that we have defined to move behaviour into a Data Class smell in an attempt to remove the smell.*

```
strategy remove-data-class clean-class (class)
  body
    apply encapsulate-fields non-private (class)
    apply remove-getters unused (class)
    apply remove-setters unused (class)
  end

strategy encapsulate-fields non-private (class)
  body
    foreach field
      satisfying
        nonprivate-field (class, field)
      loop
        compute-getter-name(field, getter-name)
        compute-setter-name(field, setter-name)
        apply encapsulate-field (field, getter-name, setter-name)
      end
    end
  end

strategy remove-getters unused (class)
  body
    foreach accessor
      satisfying getter-method(class, accessor)
    loop
      try remove-method (accessor)
    end
  end

strategy remove-setters unused (class)
  body
    foreach accessor
      satisfying setter-method(class, accessor)
    loop
      try remove-method (accessor)
    end
  end
```

Listing 6.12: *Strategies that we have defined to clean a Data Class smell by encapsulating its fields and getting rid of its unused accessor methods.*

6.2 A refactoring planner prototype

We have assembled a prototype that integrates a set of tools from other authors: JTRANSFORMER, JSHOP2 and IPLASMA. It should be noticed that our intention has been to demonstrate the approach presented in this Thesis and not to build a production tool. Therefore, in some cases, the ease of integration for experimentation purposes, and the availability of a tool have been the criteria we have followed when selecting a particular tool version among their different variants.

6.2.1 Tools used in our prototype

JTRANSFORMER

Description: JTRANSFORMER is a tool developed by the ROOTS group [JTr]. It is a Java source code analysis and transformation tool based in PROLOG. It converts ECLIPSE projects into logical representations of their ASTs. These first-order-logic predicates are called program element facts (PEFs) by the authors. With this tool an ECLIPSE project can be analysed and manipulated with complex queries and transformations written in PROLOG.

Installation and versions: The tool is distributed and deployed as an ECLIPSE plugin and, in its most recent version –2.9.0–, it supports the full Java 1.4, 1.5 and 1.6 specifications, except for generics. We use the 2.3.1 version of the tool, which is not its latest version, because it was the version available when we started writing refactoring specifications and system queries. The PEF specification has changed between versions and therefore, in order to use the latest JTRANSFORMER version, our HTN domain should be migrated to the latest specification too.

Inputs and outputs: Being an ECLIPSE plugin, it uses a project from the IDE’s workspace as its input. The tool translates the project’s source code into a PEF representation and offers a PROLOG console to interact with the PEF factbase and a tool to explore and inspect it. The tool produces two kinds of output: to an ECLIPSE project and to a file. The system’s source code and the factbase are kept in sync, so changes in any of the system’s representations are propagated to the other. In addition to this, the tool can export the current system’s factbase representation to a text file. This file contains the set of PEFs representing the whole system’s source code, as well as the additional PEFs representing the binary elements referenced from the source code’s AST, such as those of the JAVA language library. PEFs are written in PROLOG syntax. For reference purposes, the Java PEFs specification we use is available at the website of the ROOTS research group [JPs].

What do we use it for?: We use JTRANSFORMER for two purposes. We use the tool to process a software system’s source code, in the form of an ECLIPSE project, obtaining an output text file that contains the logic-based representation of it in the form of a PEF factbase. We also use it to write, test and debug system queries. We maintain a separate ECLIPSE project with system queries written in PROLOG and we use the PROLOG console of JTRANSFORMER to test them.

Limitations: The JTRANSFORMER version we use presents some limitations that restrict the range of systems that can be processed with it. The ECLIPSE project used as the JTRANSFORMER input should compile correctly. This means that all references to, for example, any external libraries used in the project, should be resolved. As far as we know, the tool considers each single project as a separate system. We can use it with different projects within the same workspace, but we have not been able to generate a joint PEF representation comprising them.

This makes it difficult to use it with big software systems, which are usually organised into several projects. The 2.3.1 version does not support generics either and, as a consequence, the tool will not accept a project with generics appearing in the source code as a valid input.

According to the authors of JTRANSFORMER, the latest version of the tool overcomes these limitations. It can deal with unresolved references. Therefore, it is able to work with project parts or in the absence of some required libraries. It does support systems which are split into different projects. It does not support generics fully yet, but it can parse and represent any system's source code, even if it uses generics.

JSHOP2

Description: JSHOP2 is an HTN forward planner developed in the Computer Science Department of the University of Maryland [JSHa]. The planner has already been described in the previous chapter (see 5.3.2), so only some additional interesting technical aspects are detailed here.

Installation and versions: The planner is a standalone tool that can be obtained in different “flavours”: SHOP2, which is the original LISP version, JSHOP2, which is the JAVA version we use, and JSHOP2GUI, which is a modified version of JSHOP2 that integrates a graphical interface. Both SHOP2 and JSHOP2 are used through command-line interfaces. We use the 1.0.3 version of the tool, which is the latest one. The graphical version displays a tree for visualizing how the search space and the task network are being explored during the planning process. We have chosen the command-line version of the tool because it is easier to integrate with other tools, as it can communicate with them through text files; it can be used for performing automated experiments, because it can be run in the background, with no need for user interaction; and because it is much more efficient than the graphical version.

Inputs and outputs: The JSHOP2 planner needs two main text files as its inputs: a planning domain and a planning problem. The planning domain file contains the methods, operators and axioms that comprise the problem domain. The planning problem file contains one or several specifications of planning problems, each one composed of the initial state of the system, as a set of ground first-order-logic predicates, and a goal, as a ground task list. These two files have to be written in the JSHOP2 language [Ilg06], which presents a LISP-like syntax. The keywords of the JSHOP2 language cannot be used in the specifications of the domain or the planning problem. This can be taken into account when writing domain specifications, but it forces us to pre-process the representation of the initial state of the system, in order to “protect” the forbidden terms that can appear in JAVA entities' names. The external user-defined procedures, that can be invoked during the planning process through call terms, should be specified as JAVA classes implementing an abstract method. These JAVA source files, one per external procedure, are additional input files complementing the domain definition.

As its output, the planner produces the sequence of operators as a list of ground primitive task symbols that constitute the found plan. Some details about the planning process are also given: the elapsed time of the process and the plan cost –the sum of all costs of the operators in the plan. The planner can be asked to produce only the first valid plan it finds, a maximum number of plans, or all the plans it can find. The plans are dumped to the standard output so it is easy to redirect and store them in a text file.

What do we use it for?: We use JSHOP2 to compute refactoring plans. We specify the refactoring planning domain as a set methods, operators, axioms and external procedures, as described in Chapter 5. This domain implements a set of system queries, refactoring strategies,

refactorings and non-behaviour preserving transformations, that are gathered in the domain definition input file. It also includes some queries that have to be implemented through JAVA files, as external procedures. We use a logic-based representation of the current version of the system source code as the initial state in the problem definition file. The goal in this file is a ground task that represents the refactoring strategy that we want to instantiate into a refactoring plan.

The output the planner produces contains operators only. In our approach, operators are used just to represent the most basic AST changes. Therefore, in order to interpret the plan at a higher-level, and to be able to identify which strategies, refactorings and non-behaviour-preserving transformations have been included in the plan, we use some auxiliary operators. These are included in the plan each time a higher-level transformation is selected. In addition to a sequence of operators representing the atomic AST changes, the auxiliary operators in the plan provide enough information, so the refactoring plan can be digested from it at different levels of detail.

Limitations: The JSHOP2 planner is case insensitive, and due to this, some information from the original source code can be lost in the planning process. Despite the terms' capitalization in the original input files, all terms are converted to lowercase during the planning process. Since the JAVA language specification defines the language as case sensitive, this information loss can introduce ambiguities in the results. This can even produce undesired results and errors, because two JAVA entities of the same type and the same name, but with different capitalization, can be mistakenly identified.

The problem solver integrated in JSHOP2 can cope with a good range of axioms, even with rather complex ones, but is limited to strict functional computations. Some additional computation mechanisms, such as the cut operator of PROLOG (“!”), are absent. Also, external procedures do not have access to the whole system state. These limitations imply that some operations, such as some metrics computations, are difficult to implement or can only be performed in a very inefficient way. A more complete logics-engine, such as the PROLOG inference engine, would be desirable.

Some technical features of the JSHOP2 planner impose certain restrictions on how the planner can be used and integrated with other tools. One of these features is the JSHOP2 language used in both input and output files. It might not be a problem for defining the refactoring planning domain, but the representation of the system's states has to be given to the planner as a set of logical terms in the same LISP-like language.

All these limitations are acceptable for a prototype. Name capitalization loss does not really lead to conflicts or ambiguities in the end. Our approach can be demonstrated even if we cannot implement certain metrics. The set of tools we use can communicate through translators and pre-processing scripts. Nevertheless, these limitations have to be solved in order to build a production tool. The development of a custom implementation of the JSHOP2 algorithm, more integrated with the other set of tools, could solve all these problems. A PROLOG version of the planner, for example, could be easily and straightforwardly connected to JTRANSFORMER which is tightly integrated with ECLIPSE already.

iPLASMA

Description: iPLASMA is a tool by the LOOSE research group [iPl, MMM⁺05]. It is an integrated environment that performs a great variety of quality analysis over Object-Oriented

systems. Among other features, it contains tools for software visualisation, metrics computation and detection of bad smells. It implements the design smell detection strategies defined in [LM06].

Installation and versions: IPLASMA is available as a standalone research tool written in JAVA. We use the 6.1 version of the tool, which is its latest version. Other manifestations of the tool exist. INFUSION is a more complete and evolved version of IPLASMA. It detects more design smells and displays more complete reports. INCODE is an alternative version of INFUSION that is distributed and deployed as an ECLIPSE plugin. Unfortunately, we have not found a way to export the lists of smells detected with either INFUSION or INCODE.

Inputs and outputs: The tool can scan a given directory in search of JAVA source code files and the libraries referenced from them. It uses a set of JAVA files as inputs, analyses them and produces on-screen graphical reports as outputs. These reports include metrics summaries [LM06, Chapter 3], graphical visualisation of the system's structure [LM06, Chapter 4] and table-like results for the system entities. These table-like reports can be customised with filters, for example to list the entities affected by a certain design smell, and with additional columns, for example to show the fully qualified name or associated metrics for each listed system entity. Finally, the table-like reports can be exported as CSV-kind files.

What do we use it for?: We use IPLASMA to pre-compute a list of the design smells present in the targeted system. This complementary information is added to the representation of the initial system's state. Knowing which design smells an entity suffers from can be useful during the planning process. For example, we would prefer to move a feature envy method to a data class instead of to a class that does not present this smell.

Limitations: It is not possible to integrate this tool, as it is, to couple its design smell detection capabilities more tightly with the correction planning our approach provides. We have not found a way to specify new detection strategies or to customize the included ones, which would be desirable.

Possible solutions for these problems can vary from implementing, in the same environment as the planner, the detection strategies from [LM06], to using similar tools, such as Cultivate [Cul, SRK07], which might be easier to integrate, or even to collaborate with the authors of INCODE in order to be able to interact with their ECLIPSE-plugin version of the tool.

6.2.2 How the tools are integrated into our prototype

Our prototype is composed of a refactoring planning domain specification, the tools from other authors that we have described in the previous section and a set of SHELL and SED¹ scripts that are used to translate the files interchanged between tools, to prepare the necessary inputs for the planner and to launch the planning experiments. A general schema of how these parts are integrated is shown in Figure 6.10.

To search for refactoring plans, we initially obtain the logic-based representation of the system we want to process. We rely on JTRANSFORMER to convert the system's AST into a set of logical terms. This set builds up the initial state of the system for the planner. Additionally, we use iPlasma to detect the design smells in the system, and to generate smell-entity reports that complement the information about the current state of the system. We generate a separate report per design smell kind. These processes are both interactive. They cannot be automated and thus, have to be performed manually. The result of these processes are several text files: one containing the set of PEFs representing the system's initial state and another set for the smell

¹The UNIX command line stream editor.

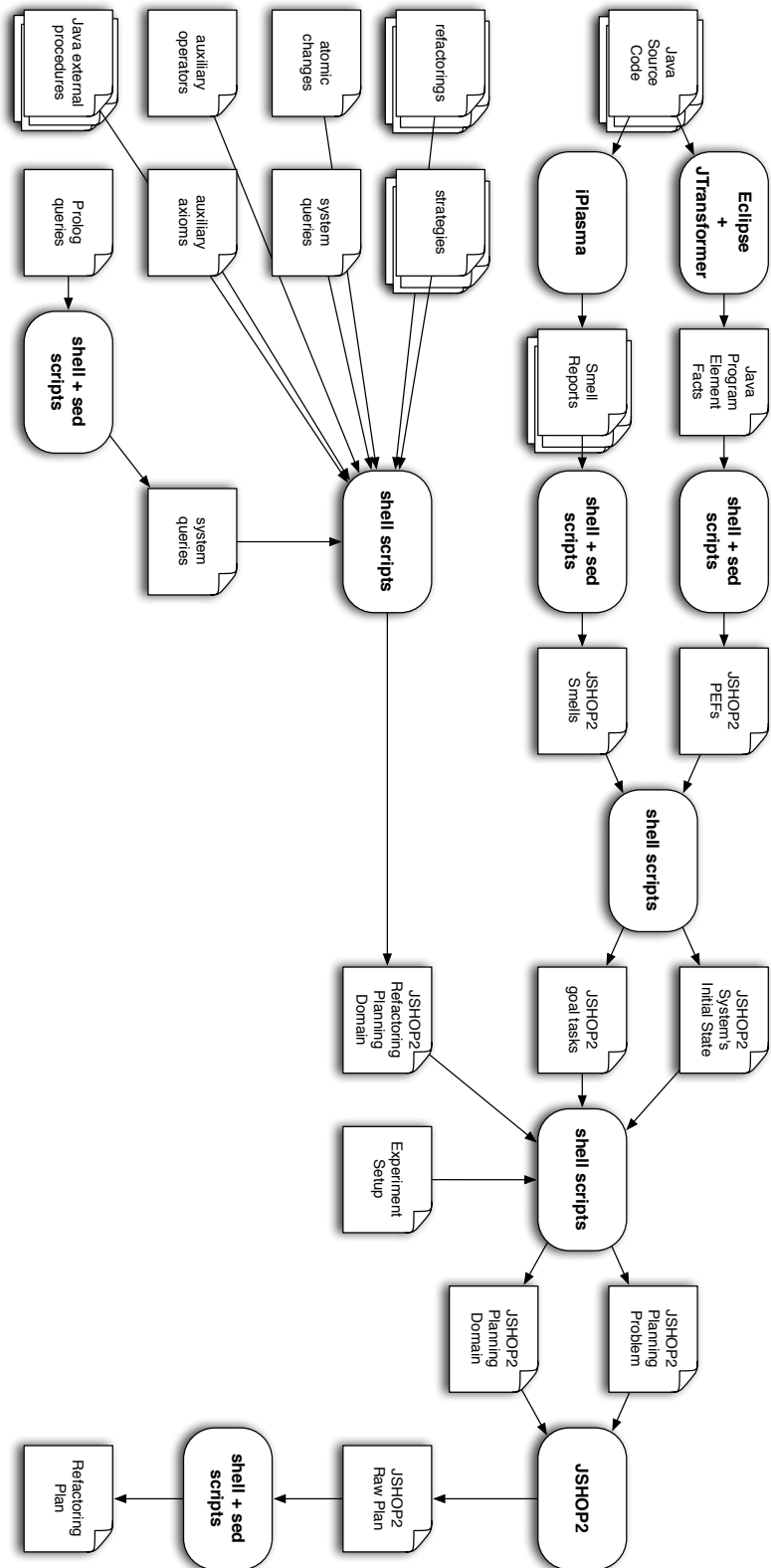


Figure 6.10: Overview of the different tools used in the experiments and how they are integrated through SHELL and SED scripts. Tools are depicted as ovals and files are depicted as dog-eared notes.

reports, in the form of lists of the affected entities with their fully qualified names. This process should be run for each software system, and moreover, for each version or state of the system we want to address, in order to obtain the initial state files.

A set of `SHELL` and `SED` scripts translates the output produced by the `JTRANSFORMER` tool, which is `PROLOG`-based, to a `LISP`-based version matching the syntax and rules of the `JSHOP2` language. The smell reports are also converted to first-order-logic predicates that relate smells and the affected entities. This translation and pre-processing stage is performed automatically when needed – the first time a certain initial source code is used. The result of this process is a file containing the specification of the system’s initial state written in the `JSHOP2` language.

We have organised the refactoring planning domain in different text files: one for each refactoring strategy and refactoring implementation, one for the operators implementing the basic AST changes, one for a set of system queries, one for a set of auxiliary methods, operators, and axioms, and a set of `JAVA` files implementing a few external procedures (See Appendix C for a reference list). The system queries are originally written and tested in `PROLOG` using `ECLIPSE` and `JTRANSFORMER`. A set of `SHELL` and `SED` scripts translates these `PROLOG` queries automatically into the `JSHOP2` language. The refactoring planning domain definition file is compiled by another set of `SHELL` scripts that gathers all the `JSHOP2` files into a single file.

A launcher `SHELL` script prepares the system’s initial state, the refactoring planning domain, etc., to use them as inputs for the planner, and then runs the computation of the requested plans by invoking the `JSHOP2` planner. The computation can be configured to request a single plan, or to launch a battery of experiments. The raw planner’s output, containing the resulting plans and some debugging information, is then collected in a text file. The produced plans are processed with `SHELL` and `SED` scripts to remove the low-level details – the sequence of basic operators in the plan – and a digested plan, containing only refactorings, and refactoring strategies, is finally produced.

6.3 Description of the case study

In order to exercise our prototype and evaluate our approach, we have performed some experiments based on case studies. We present two case studies that address the instantiation of refactoring strategies for removing the *Data Class* and *Feature Envy* design smells. They use the refactoring planning domain described in Section 6.1.

6.3.1 Experiments setup

Description and objectives

The evaluation of our approach has been performed in an empirical way and has focused on the efficiency and scalability of the planning process. The effectiveness of our initial refactoring planning domain has also been tested. Nevertheless, the ability of our prototype to produce refactoring plans that effectively correct a design smell is linked to the range of situations covered by the strategies defined in the refactoring planning domain. Since the number of cases addressed by our sample refactoring planning domain is rather limited, we have given more attention to the efficiency and the scalability of the approach rather than to the effectiveness of the plans. Therefore, the experiments presented here are mainly oriented to testing the time consumed by the process regarding the size of the targeted software system. Using the template for experiment definition from [WRH⁺00], our intention with this case study can be described as:

Experiment Goal: Analyse our refactoring planning approach for the purpose of characterisation and evaluation with respect to their effectiveness, efficiency and scalability from the point of view of the researcher in the context of the reference prototype we have assembled.

The objectives of the experiments are further decomposed and listed below:

To characterise our prototype. The first objective of the experiments is to give an overview of the efficiency of our approach and of the reference prototype we have assembled.

To evaluate the scalability of the approach. One of our major concerns is the feasibility of applying our approach over real systems, in a size range from medium to large. We aim to check whether our approach could be implemented as a production tool that would produce refactoring plans in a reasonable time.

To establish a base line for future research. To the best of our knowledge, we do not know of any other approach dealing with automated refactoring planning. Therefore, it is necessary to measure our prototype behaviour so that future research and improvements can be compared against it.

Variables

The variables used in this study are listed below:

- Independent variables:
 - *PEF*: number of program element facts in the initial state of a software system representation. Defines the size of a system and therefore the size of each experiment.
 - *FE*: number of *Feature Envy* design smells in the initial state of a software system representation. Defines the number of Feature Envy experiments to be run for each system.
 - *DC*: number of *Data Class* design smells in the initial state of a software system representation. Defines the number of Data Class experiments to be run for each system.
- Dependant variables
 - *P*: Number of plans obtained for each system. It is used for characterising the effectiveness of the approach.
 - *T_t*: Total elapsed time. Time elapsed for running an experiment completely, from launching the JSHOP2 tool until the execution of the planner ends. Total elapsed time is the sum of precompilation time and planning time. It is used for characterising the efficiency and scalability of the approach. This variable is measured in seconds.
 - *T_c*: Precompilation time. Time elapsed during the precompilation time of the JSHOP2 tool, from launching the JSHOP2 tool until starting the generated planner to initiate the planning process. It is used for characterising the efficiency and scalability of the approach. This variable is measured in seconds.
 - *T_p*: Planning time. Time elapsed during the planning process, from starting the planner until the termination of the planning process. It is used for characterising the efficiency and scalability of the approach. This variable is measured in seconds.

	System	Version	NOC	NOM	LOC	PEF	Feature Envy	Data Class
0	Void	—	1	1	4	12696	—	—
1	JTombstone	1.1.1	40	233	1938	32780	2 (2)	7
2	Groom	1.3	34	243	3699	35434	2 (5)	2
3	Lucene	1.9	199	1998	17627	85064	18 (25)	16
4	Pounder	0.96	218	1318	9410	98570	26 (26)	3
5	MyTelly	1.2	72	941	12625	133605	2 (6)	13
6	Jwebap	0.6.1	114	1278	16417	141047	17 (18)	21
7	dbXML	2.0	389	3336	25862	199400	21 (30)	40
8	GanttProject	2.0.10	515	5048	40775	241095	36 (64)	27
9	JfreeChart	1.0.11	561	8024	80668	354543	45 (46)	29
Total number of experiments							169	158

Table 6.1: Details of the system used in our experiments, with its number of classes (NOC), methods (NOM), lines of code (LOC), program element facts in the first-order logic representation of the system (PEFs), Feature Envy and Data Class design smells. The first row –Void– represents, as a reference, an empty JAVA system.

Samples

In order to test our approach we ran experiments over a set of open source systems. The subjects of our study are: JTOMBSTONE, a program for reporting dead code in Java programs [JTo]; GROOM, a lightweight webserver [Gro]; LUCENE, a textual search engine for Apache [Apa]; POUNDER, a GUI testing utility [Pou]; MYTELly, a program for managing custom TV listings [MyT]; JWEBAP, a profiler tool for monitoring Java systems [Jwe]; DBXML, a native XML database [dbX]; GANTTPROJECT, a project scheduling and management tool [Gan]; and JFREECHART, a data chart library [JFr].

As mentioned in Section 6.2, the limitations of some of the tools we use restrict the systems that can be processed with our prototype. These limitations have been the main criteria we have followed to select the systems. We have browsed sourceforge manually and searched among those systems which are suitable to be used with our prototype. We have gathered a set of systems of different sizes, which vary from small to medium in size.

Table 6.1 summarises the details of the targeted systems, which have been obtained by analysing them with iPlasma. The table shows the number of classes, methods (NOM), lines of code (LOC), *Feature Envy* and *Data Class* smells detected by the tool. We have removed some smells from the experiments because they affected structures our representation does not support, such as nested classes. The total number of smells detected by iPlasma is shown between parentheses. The number of smells we have used for our experiments is shown without parentheses. It is worth noticing that an additional entry, labelled “Void”, has been included as the first row of the table. This entry represents a JAVA system composed of a single class, enclosed in the default package, and that only contains a single and empty main method. It serves as a reference to illustrate that a part of the initial state factbase belongs to the basic definitions of the JAVA language. The table is ordered by factbase size because the factbase representation of each system is what is used as the actual input for the planning process.

The materials, data and results of this case study are available at <http://www.infor.uva.es/~jperez/thesis>.

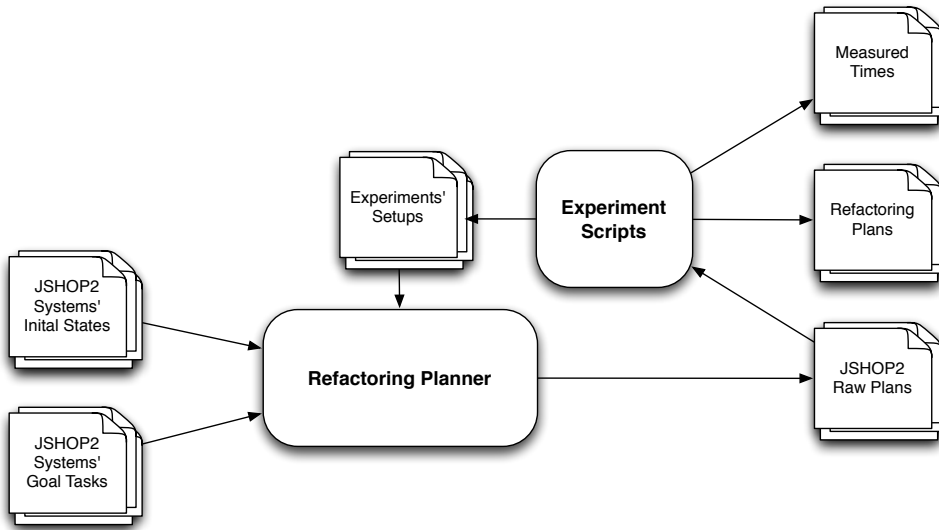


Figure 6.11: *Overview of the experiment's execution environment.*

Execution environment

The experiments' execution environment is comprised of a set of experiment scripts. These scripts prepare each experiment setup, and launch the refactoring planner described in Section 6.2.2.

For each experiment, we have collected the total elapsed time (T_t), the planning time (T_p) and the raw output of the planner. The planning time (T_p) is measured by JSHOP2 and included in its output, the total elapsed time (T_t) is measured by the experiment scripts and the precompilation time (T_c) is computed as $T_t - T_p$.

As described in Section 6.2.2, the raw output of the planner produces not only the plan but also a good amount of debugging information. This output has been filtered afterwards in order to extract the relevant information regarding the sequence of refactorings in the plan and the strategy paths traversed through the task network to obtain them.

At the end of each run, the experiment scripts process the raw output of the planner in order to obtain the measured times and the digested refactoring plans for each experiment. An overview of this setup is shown in Figure 6.11 (see also Figure 6.10 for the overview of the refactoring planner prototype).

The experiments have been executed on a Core 2 Quad Q9400 / 2.66 GHz computer with 4GB of RAM running Ubuntu Linux 9.10 operating system.

Experiment procedure

The experiments comprise the execution of the refactoring planner prototype for solving a set of 324 refactoring planning problems, aimed at removing *Feature Envy* and *Data Class* smells – 169 and 158 cases –, which span across the 9 software systems selected. Each experiment requests the planner to search for a plan as a solution for instantiating one of the refactoring strategies previously described in 6.1.2. Each experiment is a planning problem whose goal is to achieve a design smell refactoring strategy. The arguments for the strategy are the entities involved in each design smell detected.

Feature Envy								
	System	PEFs	Smells	Plans	%	Mean T_t	Mean T_c	Mean T_p
1	Jtombstone	32780	2	1	50	29.52	23.02	6.49
2	Groom	35434	2	1	50	30.55	21.57	8.98
3	Lucene	85064	18	3	16.67	64.1	53.71	10.4
4	Pounder	98570	26	21	80.77	107.7	103.66	4.04
5	MyTelly	133605	2	0	0	253.69	182.85	70.84
6	Jwebap	141047	17	9	52.94	182.03	148.05	33.98
7	dbXML	199400	21	11	52.38	413.21	303.29	109.92
8	GanttProject	241095	36	13	36.11	544.79	385.46	159.32
9	JfreeChart	354543	45	24	53.33	1065.54	960.83	104.71
Totals			169	83	49.11			

Data Class								
	System	PEFs	Smells	Plans	%	Mean T_t	Mean T_c	Mean T_p
1	Jtombstone	32780	7	6	85.71	26.28	22.5	3.78
2	Groom	35434	2	2	100	30.56	21.63	8.93
3	Lucene	85064	16	16	100	62.96	53.48	9.48
4	Pounder	98570	3	3	100	179.91	104.28	75.63
5	MyTelly	133605	13	11	84.62	206.36	186.15	20.21
6	Jwebap	141047	21	20	95.24	167.55	148.36	19.2
7	dbXML	199400	40	40	100	431.47	300.36	131.11
8	GanttProject	241095	27	24	88.89	530.18	386.57	143.62
9	JfreeChart	354543	29	23	79.31	1021.41	959.76	61.65
Totals			158	145	91.77			

Table 6.2: Summary of the results regarding the number of produced plans and the elapsed times of the planning process. Times are given in seconds.

The experiments have been prepared according to the prototype described in Section 6.2.2. The initial state of each system, comprised of its first-order logic representation and the smell-entity predicates with the design smells detected, have been prepared for all samples prior to the execution of the experiments. The experiments have been run in batch mode and the resulting files: raw plans, refactoring plans and measured times, have been collected afterwards.

6.4 Analysis of the results

The summary of the results obtained are shown in Table 6.2. This table displays an initial overview of the prototype. These results are discussed in further detail in this section. The analysis of the results will be focused first on the produced plans, and then on the efficiency and the scalability of the approach.

6.4.1 Discussion on the produced plans

The first point worth discussing is the difference in the total number of plans produced for each design smell. In the case of the *Feature Envy* case study, our prototype has been able to produce almost 50% of the requested plans, while it has generated plans for over 90% of the experiments in the *Data Class* case study. Despite the simplicity of the sample strategies implemented in the prototype, the results obtained are quite acceptable. Therefore, we consider them as good results. The difference in the success ratios between the *Feature Envy* and the *Data Class* case studies is due to the nature of the implemented strategies. The *Feature Envy* strategy can only generate a

Feature Envy											
	System Plans	1	2	3	4	5	6	7	8	9	Totals
		1	1	3	21	0	9	11	13	24	83
Trivial (Remove Method) St.		0	0	2	19	0	1	7	6	7	42
Move Method St.		1	1	1	2	0	8	4	7	17	41
To Data Class St.		0	1	1	0	0	5	1	1	5	14
To Envied Class St.		1	0	0	2	0	3	3	6	12	27

Data Class											
	System Plans	1	2	3	4	5	6	7	8	9	Totals
		6	2	16	3	11	20	40	24	23	145
Trivial (Remove Class) St.		1	0	0	0	3	0	22	4	1	31
Reorganize Class St.		5	2	16	3	8	20	18	20	22	114
Move Feature Envy Methods St.		0	1	1	0	0	2	1	1	3	9
Move Client Methods St.		2	2	3	2	4	2	4	8	10	37
Clean Class St.		5	2	16	2	8	20	18	17	16	104

Table 6.3: Summary of the strategies traversed for producing the plans per system and strategy type.

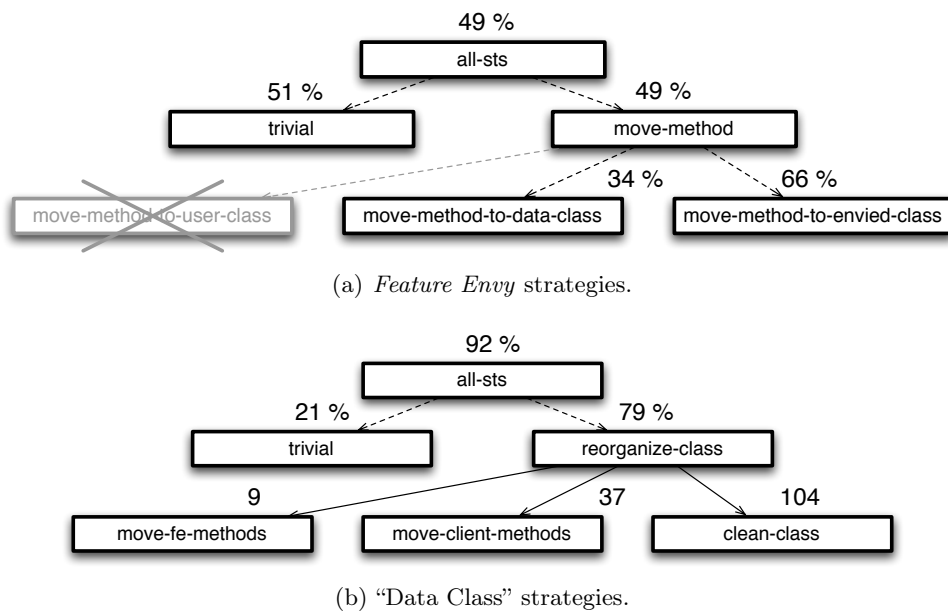


Figure 6.12: Distribution of the strategy "paths" traversed for computing the plans obtained.

plan when the targeted method can be either removed or displaced to another class, closer to the data it accesses. The *Data Class* strategy has more chances of success. It can generate a plan when the targeted class can be removed, when any method can be transferred to the class, or when some “class cleaning” is performed by improving the encapsulation of its publicly exposed fields. Further efforts in the refactoring planning domain and smell strategies could include additional knowledge to improve these figures. The distribution of how the substrategies have been traversed to obtain the resulting plans is summarised in Table 6.3, and graphically shown, in a more concise manner, in Figure 6.12. Dashed arrows represent alternative substrategies and are decomposed by percentages. Regular arrows represent complementary strategies and therefore, just a count is shown. The top strategies display the ratio of plans obtained for each design smell from the total number of experiments launched. The strategy that asks the user for the target class to host the moved method has been disabled in the experiments, since they are meant to be run unattended.

Table 6.4 summarises the number of refactorings in the produced plans per system and refactoring type. A graphical depiction of the number of refactorings in the produced plans per case study is shown in Figure 6.13. Some observations can be extracted from these results. In the *Feature Envy* case, the majority of method movements have been done by completely moving the method. Only in a few cases, 5 against 36, the method had to be moved by leaving a delegating method behind, in the original method’s place. Most *Feature Envy* smells in system 4 –POUNDER– have been corrected by removing the affected methods. This is because this system is a test utility for JAVA GUIs. A closer examination of these methods reveals that they are not actually used within the system, but they seem to expose an API for offering the system’s test functionality. It should be noticed that our approach does not discriminate whether the cases programmed for planning are actually design smells or not, so this inspection and the consequent decision should be performed by the developer, either before or after requesting the refactoring plan.

Regarding the *Data Class* cases (see Table 6.4), we have found the results obtained from system 4 –POUNDER– and system 7 –DBXML– to be particularly interesting. Among the **MOVE METHOD** count in the plans of system 4, 24 out of the 27 methods displayed in the table have been displaced to one particular *Data Class*: `PounderController` in package `com.mtp.pounder.controller`. This class contains over 20 fields, exposed through accessor methods, and serves merely as a container for them. As a consequence, the planner has found many candidate methods, those accessing the class’ fields, to be moved into the class.

In system 7 –DBXML–, the planner has found that almost half the *Data Class* smells can be easily addressed by applying the **REMOVE CLASS** refactoring in order to get rid of the affected class (See Table 6.4). We have analysed these particular cases in more detail and found that 15 out of the 22 classes proposed to be removed come from just two packages: `com.dbxml.db.admin.nodes` and `com.dbxml.db.enterprise.jsp`. These classes are not used, and therefore, they are not being referenced from within the system. As in the case of the removed methods from the *Feature Envy* case study, we have tried to guess whether it could make sense to keep these classes in the system, even when they are not being used. We couldn’t find a reason for this. Therefore, we have considered it would be safe to remove them as the computed plans instructed us to do.

Regarding the **MOVE METHOD** refactorings, in the *Data Class* case, similarly to the *Feature Envy* case, the majority of the moved methods were completely moved, without the need to add a delegating method.

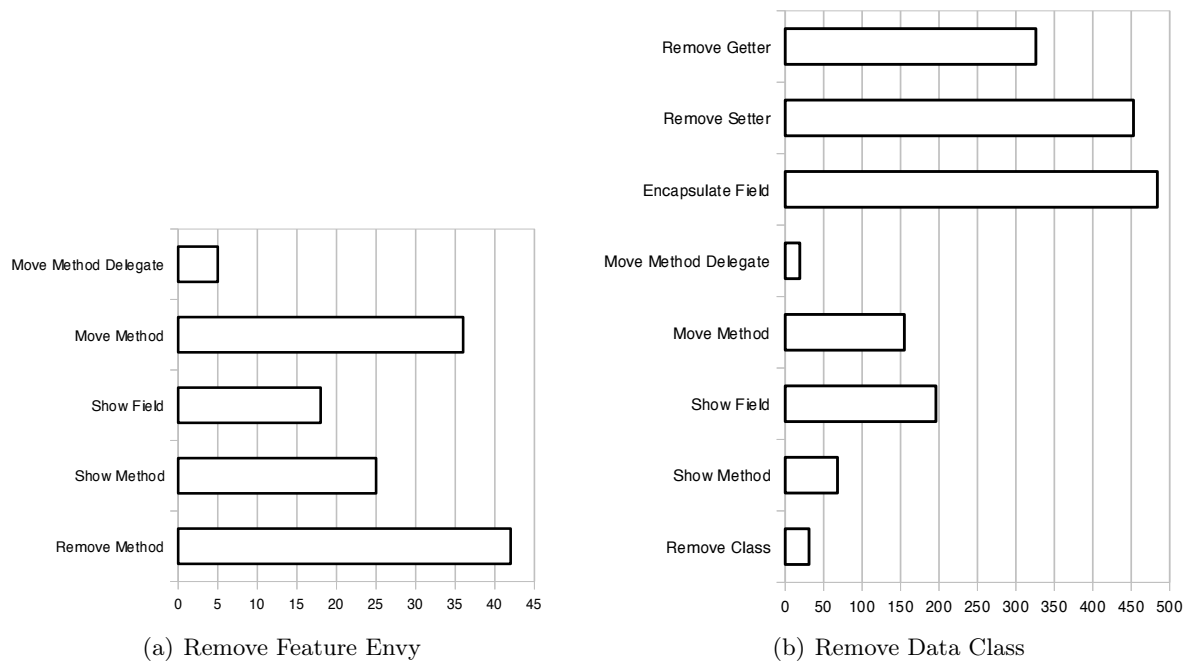


Figure 6.13: Number of refactorings and transformations in the produced plans.

Feature Env											
	Systems	1	2	3	4	5	6	7	8	9	Total
	Plans	1	1	3	21	0	9	11	13	24	83
Remove Method		0	0	2	19	0	1	7	6	7	42
Show Method		2	0	0	0	0	9	2	3	9	25
Show Field		0	1	3	0	0	6	2	0	6	18
Move Method		1	1	1	2	0	8	3	5	15	36
Move Method Keeping Delegate		0	0	0	0	0	0	1	2	2	5
Total		3	2	6	21	0	24	15	16	39	126

Data Class											
	Systems	1	2	3	4	5	6	7	8	9	Total
	Plans	7	2	16	3	13	21	40	27	29	158
Remove Class		1	0	0	0	3	0	22	4	1	31
Show Method		6	1	9	0	0	8	5	22	17	68
Show Field		10	3	29	17	0	25	10	34	68	196
Move Method		6	5	10	27	1	24	6	26	50	155
Move Method Keeping Delegate		1	0	1	0	0	6	0	7	4	19
Encapsulte Field		12	22	77	8	27	82	169	64	23	484
Remove Setter		18	13	50	8	17	58	179	42	68	453
Remove Getter		8	3	23	0	14	35	143	28	72	326
Total		62	47	199	60	62	238	534	227	303	1732
Grand Total		65	49	205	81	62	262	549	243	342	1858

Table 6.4: Summary of the plans produced and the number of refactorings in them per system and refactoring.

As for **ENCAPSULATE FIELD**, which is the most commonly used refactoring for removing a *Data Class* case in our experiment, we have noticed some smell manifestations that concentrate most of these refactorings. In one particular manifestation of the design smell, in system 7, the planner has found a *Data Class* exposing its content through over 40 fields with default `-package-` visibility. This class, `PreferencesDialog` in package `com.dbxml.db.admin.dialogs`, is a GUI class representing a dialog. The exposed fields store all the descendant graphical components contained within it. For this particular *Data Class* manifestation, the planner has generated a plan with 43 **ENCAPSULATE FIELD** refactorings (out of the 169 displayed in the table) and as many **REMOVE GETTER** and **REMOVE SETTER** refactorings as well. This is caused by a “collateral effect” of the **ENCAPSULATE FIELD** refactoring, which creates accessors when they do not previously exist. As these newly created accessors are never accessed from anywhere in the system, they are removed later on. Another 6 GUI classes, also in system 7, present the same problem as this one, with a number of **ENCAPSULATE FIELD** refactorings between 14 and 18 per *Data Class*. As a summary, all the set of GUI classes in system 7, adds up to 132 **ENCAPSULATE FIELD** refactorings, out of the 169 displayed in the table, and a similar number of **REMOVE GETTER** and **REMOVE SETTER** refactorings.

Despite the simplicity of the strategies implemented in the current refactoring planning domain, the experiments presented illustrate the benefits we claimed our approach provides. The refactoring strategy concept allows us to describe smell correction and refactoring application “recipes” in a semi-formal way and to organize them into reusable knowledge. In our experiments, the “apply move method” strategies have been used in both smell strategies developed. A substrategy of the *Feature Envy* has also been employed in the *Data Class* strategy. How the HTN planning approach, and in particular the JSHOP2 planner, provides support for instantiating refactoring strategies for each particular case has also been demonstrated. The next section will further analyse the suitability of this type of planner by discussing the efficiency of the planning process.

6.4.2 Discussion on the prototype efficiency

The discussion on the efficiency of our prototype is focused on the time elapsed for computing the refactoring plans. Table 6.2 shows an initial overview of the total elapsed mean times. They have been summarised by system and divided into their components: precompilation time and planning time.

Descriptive analysis

A quick skim over the results may show that the elapsed total time seems to be related to system size, and that bigger systems imply higher total times. Nevertheless, this has to be examined in more detail.

During the development of the refactoring planning domain definitions, we have noticed that, with JSHOP2, the biggest fraction of the total elapsed time is due to the precompilation stage. This stage involves parsing a huge domain specification, including HTNs, queries and system state, and generating a JAVA program with all the problem definition embedded as methods, strings, arrays of strings, variables etc., within its source code. This process, as stated by the authors of JSHOP2, performs some optimizations while synthesizing a problem-specific planner that may increase the efficiency of the planning process [IN03]. Nevertheless, the authors use problem domains with a small number of predicates in their initial system’s state defini-

	Feature Envy	Data Class
Size / Mean total elapsed time	0.97	0.98
Size / Mean precompilation time	0.95	0.95
Size / Mean planning time	0.82	0.43

Table 6.5: *Pearson’s correlation coefficients between PEFs (system size) and mean elapsed times for each design smell.*

tion. Their biggest examples contain about 600 predicates, while our experiments range from 32780 to 354543. We suspect that the precompilation stage is very inefficient for problems with initial states of this size. We even had to implement minor modifications to the JSHOP2 pre-compilation code because the original did not support our larger initial system states and often crashed. Our modifications allowed the pre-compilation stage’s planner synthesizer to run considerably faster. We speculate that the benefits of the precompilation stage may not apply to our case. The added cost of the precompilation stage, which is significant for problems like ours, might overcome the improved efficiency of the planning process. We intend to explore this further in future work.

We have computed the Pearson’s correlation coefficients between PEFs (system size) and mean elapsed times for each design smell. The correlation coefficients, which are displayed in Table 6.5, show that the tight correlation between system size and total elapsed time might be due to the correlation between system size and precompilation time. The correlation between size and planning time is not so evident, it is lower in the *Feature Envy* case and much lower in the *Data Class* case. As will be shown in the normality tests we have performed, the planning time variable does not present a normal distribution and therefore this correlation analysis has only a purely descriptive interest. Inferential analysis of the dependency between system size and planning time is carried out later on in this section. Nevertheless, this quick analysis serves to warn us that precompilation and planning times have to be examined separately.

As a consequence of all these initial considerations, and in order to give more precise results, we have decided to measure both the precompilation time and the planning time, and to examine them separately. Hence, our efficiency analysis is more focused on the planning time costs. As summarised in Tables 6.2 and 6.5, not only the mean precompilation times are higher than the mean planning times, but in addition, precompilation times seem to be tightly correlated to system size. Regarding the planning times, as the targeted system gets bigger, the planning time also seems to get larger, but the correlation with system size is less clear than in the case of precompilation time.

Therefore, the potential threats to the scalability of our approach, if any, might be mainly caused by the JSHOP2 pre-compilation stage. This may be improved in the future by implementing a custom planner, based on JSHOP2, that would avoid this overload. Moreover, even if the growth of the mean planning time were to be correlated to the targeted system size, it appears to scale for even bigger systems. The mean planning time is under 3 minutes in all cases. Therefore, we consider this result to be quite acceptable for our prototype in terms of efficiency and scalability. The great benefit of the HTN planning approach is that the planner searches a severely pruned state space. The time results are quite good despite the refactoring planning problem having a huge search space.

As for the differences between both design smell case studies (see Table 6.2), for some systems, the mean planning times do not seem to differ very much between the two explored case studies

–*Data Class* and *Feature Envy*. At first sight, the mean planning time is higher in the case of *Feature Envy* experiments for systems 1, 5 and 6. In the case of the *Data Class* experiments, they seem to be higher for systems 4, 7 and 9. Finally, the mean planning times for systems 2, 3 and 8 appear to be rather similar between both case studies. Therefore, there is no clear evidence of the planning times being generally dependent on the requested strategy.

In order to analyse precompilation and planning time in more detail, their raw figures are graphically depicted, separated per design smell case study, in Figures 6.14 and 6.15 as scatter-plots. These diagrams clearly reveal that the precompilation time (see 6.14(a) and 6.15(a)) is almost the same regardless of the individual experiments and the case study, and it seems to be only dependent on the system’s state size. These Figures also illustrate that the majority of planning times (see 6.14(b) and 6.17(b)) actually fall below the 3 minute limit. In addition, they also reveal that the planning time results are very dispersed. It also seems that this dispersion may increase alongside system size.

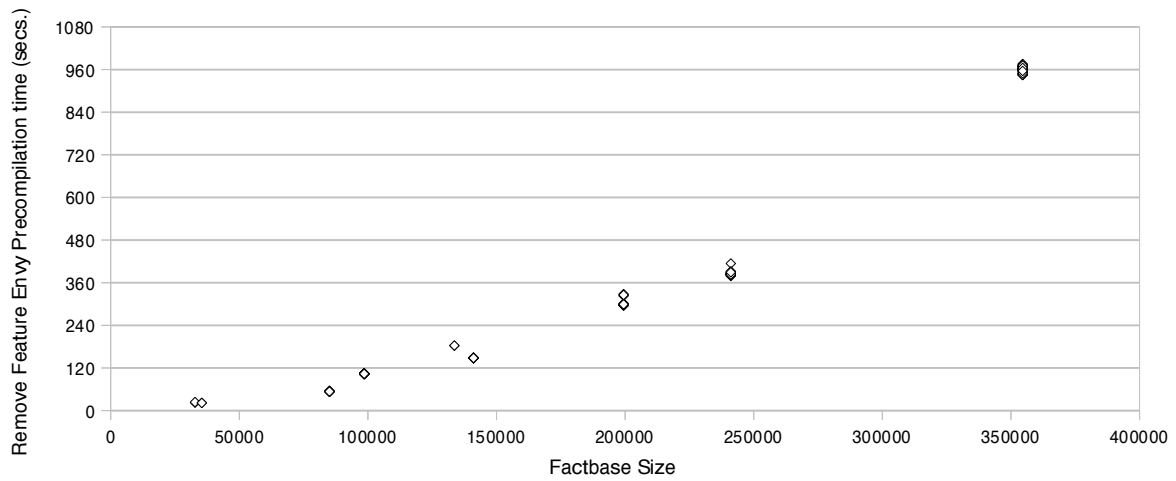
From this point, the discussion will be focused on the planning time. This will be reviewed in greater detail.

Review of the outliers

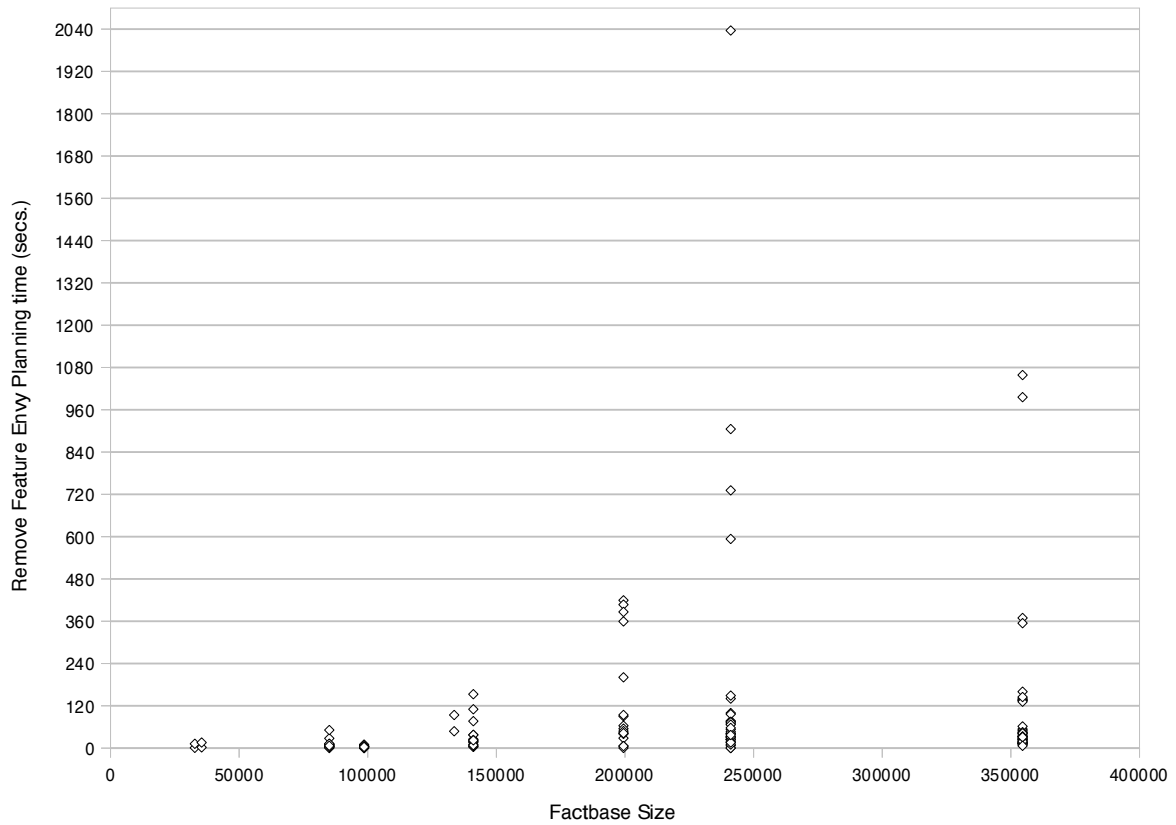
A closer description of the dispersion of the planning times has been performed by obtaining additional statistics and by representing them with boxplot diagrams. The computed statistics are compiled in Tables 6.6 and 6.7. Boxplots representing the distribution and dispersion of the planning time are shown in Figure 6.16. In order to find the cause of this dispersion, we have performed a closer examination of the outliers. We have analysed the outliers in the bigger systems –dbXML, GanttProject and JFreeChart– but we could not find any general cause for them.

We have reviewed whether the higher planning times are due to the planner failing to find a plan. In these cases, the planner would presumably search a bigger state space until running out of possible paths. Therefore, it might make sense that these experiments would take longer to compute. Nevertheless, this cannot be rightfully confirmed. We have looked more carefully into the *Feature Envy* experiments’ results because the number of successful and unsuccessful plan computations in this case study were balanced. In the experiments of system 8 –GANTTPROJECT–, with 4 outliers, the three highest planning times belong to unsuccessful plan computations while the fourth one belongs to a successful experiment that produced a valid plan. In system 9 –JFREECHART– experiments, with 9 outliers, the two most extreme ones belong to experiments for which the plan was not found, but the other 7 outliers belong to experiments for which valid plans were successfully obtained. However, in system 7 –DBXML– experiments, with 4 outliers, the two highest planning times belong to successful plan computations, while the other two belong to unsuccessful ones. We did not see that the dispersion and the higher planning times could be clearly and entirely related to not finding a plan, at least with the data we have.

We have also analysed the outliers of the *Data Class* study case. All the outliers we have reviewed belong to experiments for which a plan has been computed successfully. Therefore, the outliers of this case study have been useless for searching for more evidence of whether not finding the plan could affect the planning time. In this case study, we have specifically searched for a relation between the higher planning times and the number of refactorings in the plan. In system 7 –DBXML– with 9 outliers, we have found that the experiment with the highest planning time corresponds to the plan with the highest number of refactorings among all the experiments –127.

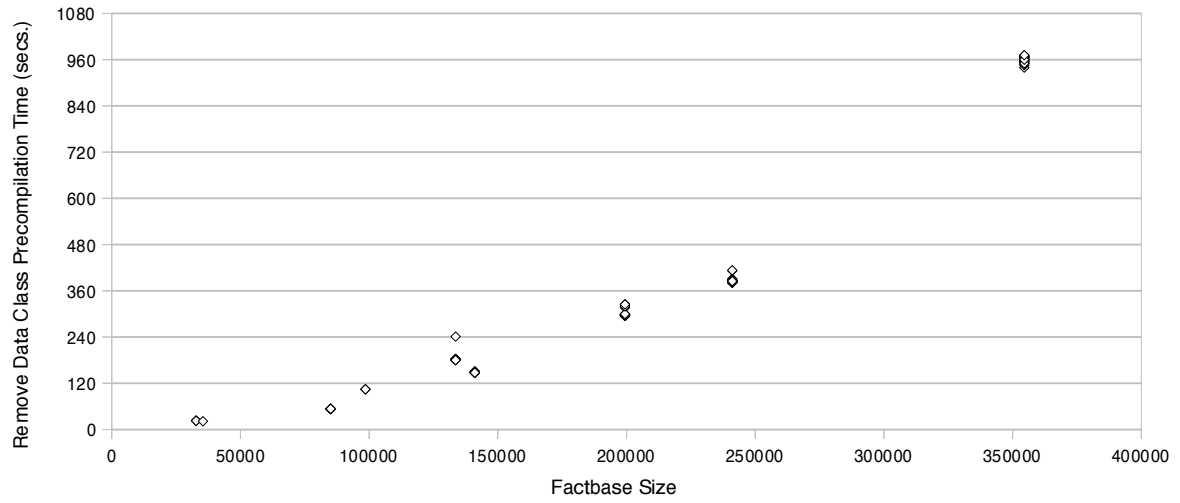


(a) Precompilation time of the “Remove Feature Envy” case study per system size.

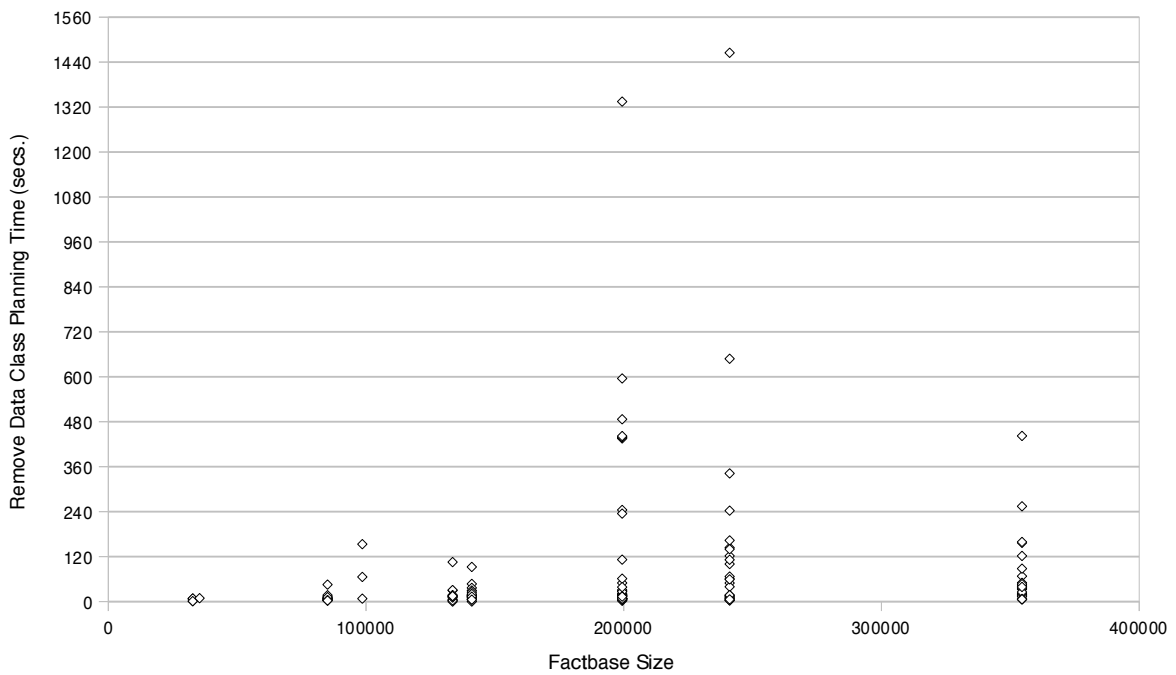


(b) Planning time of the “Remove Feature Envy” case study per system size.

Figure 6.14: Distribution of the elapsed time for the Feature Envy experiments per system. The total time is presented in two separate and well distinguished stages: the JSHOP2 precompilation stage time and the actual planning time. Time is given in seconds. System size is given as the number of predicates in the factbase representing the software system.



(a) Precompilation time of the “Remove Data Class” case study per system size.



(b) Planning time of the “Remove Data Class” case study per system size.

Figure 6.15: Distribution of the elapsed times for the Data Class experiments per system. The total time is presented in two separate and well distinguished stages: the JSHOP2 precompilation stage time and the actual planning time. Time is given in seconds. System size is given as the number of predicates in the factbase representing the software system.

Feature Envy								
System	Plans	Mean Plan. Time	Median	Std. Deviation	Min.	Max.	Range	Coef. of Variance
1	2	6.49	6.49	8.11	0.76	12.23	11.47	124.93%
2	2	8.98	8.98	9.69	2.13	15.83	13.70	107.89%
3	18	10.40	8.20	11.81	0.10	51.23	51.13	113.56%
4	26	4.04	3.64	2.00	0.05	10.01	9.96	49.53%
5	2	70.84	70.84	32.60	47.79	93.89	46.10	46.01%
6	17	33.98	19.89	41.50	3.44	152.98	149.54	122.13%
7	21	109.92	42.83	148.23	0.43	419.37	418.94	134.86%
8	36	159.32	40.47	379.76	0.06	2036.04	2035.99	238.36%
9	45	104.71	32.62	216.00	6.54	1058.49	1051.95	206.29%

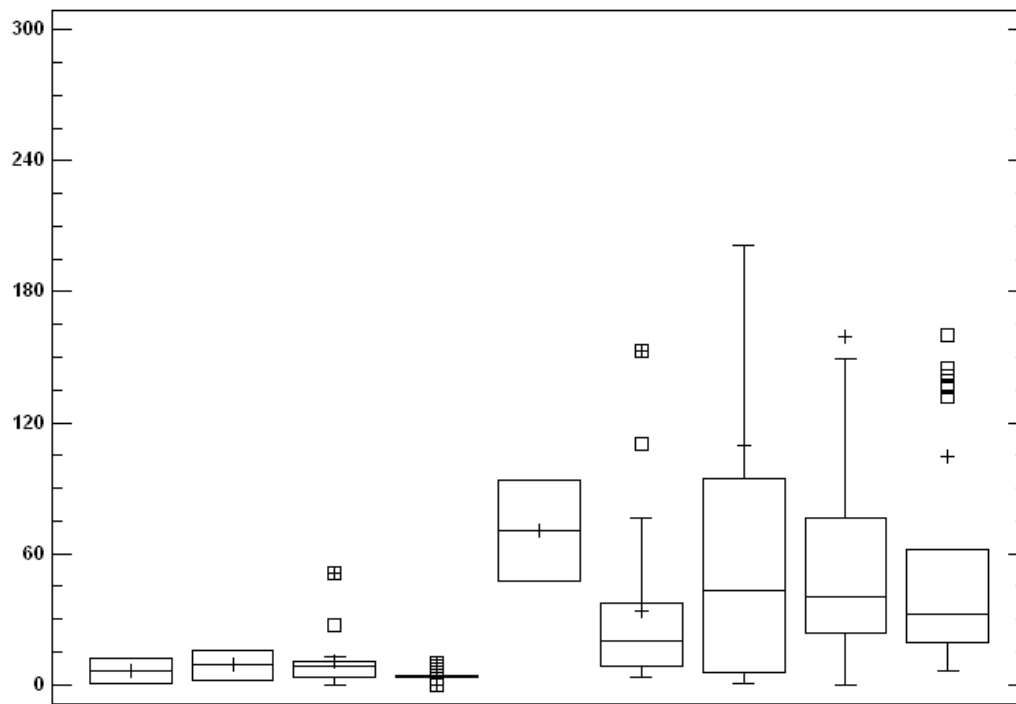
Data Class								
System	Plans	Mean Plan. Time	Median	Std. Deviation	Min.	Max.	Range	Coef. of Variance
1	7	3.78	1.57	3.58	0.26	8.34	8.08	94.92%
2	2	8.93	8.93	0.05	8.89	8.96	0.07	0.56%
3	16	9.48	7.24	10.42	2.56	45.30	42.74	109.87%
4	3	75.63	65.80	73.19	7.86	153.24	145.38	96.77%
5	13	20.21	13.59	27.23	0.00	105.35	105.35	134.72%
6	21	19.20	14.84	20.28	0.01	92.45	92.45	105.66%
7	40	131.11	16.43	258.01	2.07	1334.63	1332.55	196.78%
8	27	143.62	49.46	297.85	2.96	1464.97	1462.00	207.39%
9	29	61.65	32.60	93.17	5.79	442.06	436.27	151.12%

Table 6.6: Descriptive statistics of the experiments' results, regarding the T_p variable, summarised by system and design smell study case.

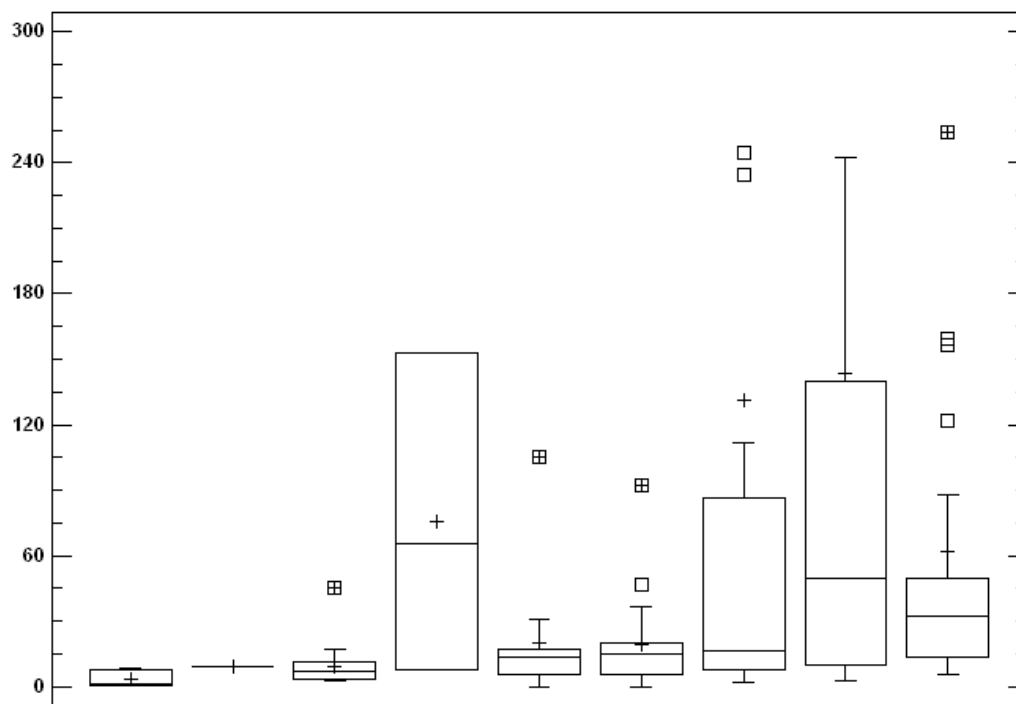
Feature Envy							
System	Q1	Q3	IQ Range	Outliers (below)	Outliers (above)	Plans	Plans – Outliers
1	0.76	12.23	11.47	0	0	2	2
2	2.13	15.83	13.70	0	0	2	2
3	3.63	10.76	7.13	0	2	18	16
4	3.43	4.35	0.92	1	3	26	22
5	47.79	93.89	46.10	0	0	2	2
6	8.39	37.52	29.13	0	2	17	15
7	5.87	94.20	88.33	0	4	21	17
8	23.88	76.04	52.17	0	4	36	32
9	19.17	61.60	42.43	0	11	45	34
Total	5.87	50.39	44.52	1	26	169	142

Data Class							
System	Q1	Q3	IQ Range	Outliers (below)	Outliers (above)	Plans	Plans – Outliers
1	0.92	7.97	7.06	0	0	7	7
2	8.89	8.96	0.07	0	0	2	2
3	3.50	11.16	7.66	0	1	16	15
4	7.86	153.24	145.38	0	0	3	3
5	5.86	17.34	11.48	0	1	13	12
6	5.92	20.10	14.18	0	2	21	19
7	7.84	86.56	78.72	0	9	40	31
8	9.86	139.76	129.90	0	3	27	24
9	13.57	49.98	36.41	0	4	29	25
Total	7.16	46.46	39.30	0	20	158	138

Table 6.7: Quartiles and ranges for analysing the planning time distribution and for drawing the boxplot diagrams. We have also included the count of the outliers that have been identified.



(a) Distribution of the “Remove Feature Envy” planning time per system size.



(b) Distribution of the “Remove Data Class” planning time per system size.

Figure 6.16: *Distribution of the planning time for the two case studies and per system. Systems are ordered by growing factbase size from left to right.*

Moreover, the plans of 7 out of 8 of the highest planning times among this system's experiments, include a big number of refactorings, over 40, and 50 in one case. Nevertheless, this does not apply to all outliers. One of the outliers among this system's experiments corresponds to a plan with a smaller number of refactorings –8. In addition to this, we have not found examples in system 7 of experiments with a big number of refactorings in their plans and regular planning times. In system 8 –GANTTPROJECT– with 3 outliers, the three experiments correspond to plans with a medium number of refactorings –19, 23, 27. However, among the experiments of this system, we can find experiments whose plans include a similar or bigger number of refactorings –31, 24, 17, 15– and which have taken a regular time to compute, below the Q3 of this system. In system 9 –JFREECHART– with 4 outliers, 3 of them belong to experiments whose plan contains a medium number of refactorings –31, 31, 22. Nevertheless, one of these outliers, specifically the second highest, belongs to an experiment with only 2 refactorings in the resulting plan. Among the other experiments of this system, we found cases with a big number of refactorings in the obtained plans –48, 35– which have nevertheless been computed in a regular time, below the Q3 of this system.

By reviewing the outliers for the planning time, we could not find an evident cause for the dispersion of the results. We have speculated that, in the *Feature Envy* case, the dispersion might be caused by the unsuccessful plan computations taking significantly longer than the successful ones. We also suspect that the higher planning times might be due to the application of the **ENCAPSULATE FIELD** refactoring in the *Data Class* case, maybe originated by an inefficient implementation of the refactoring. There is a big number of this type of refactorings in the plans of most of the experiments that present the highest planning times. However, we are afraid that more data is needed to confirm these hypotheses in the future.

The next sections are dedicated to performing several inferential statistical analyses over the planning time variable.

Inferential analysis of planning time.

To begin with, a quick look at the descriptive analysis of the experiments' elapsed time suggests that planning time does not follow a normal distribution. Therefore, we have performed normality tests in order to decide which statistical analysis would be the most appropriate to apply. We have split the planning time samples by system and case study. To check normality, we have performed the Shapiro-Wilk's test [SCJ88]. We have been able to apply the test to all but 5 samples, for which there were too few values –less than 5 experiments.

Test: Normality test. Shapiro-Wilk.
Variable: T_p (Planning time)
Hypotheses: H_0 : T_p belongs to a normal distribution.
 H_1 : T_p does not belong to a normal distribution.

According to the test results, which are shown in Table 6.8, we can reject H_0 with 5% significance level in all cases, therefore T (planning time) does not belong to normal distributions. As a consequence, further data tests are performed, from now on, with non-parametric analysis.

System	Feature Envy				Data Class			
	Sample Size	W	$p - value$	Reject?	Sample Size	W	$p - value$	Reject?
1	2	—	—	—	7	0.804977	4.58401E-02	yes
2	2	—	—	—	2	—	—	—
3	18	0.649539	6.40190E-06	yes	16	0.623839	1.10907E-05	yes
4	26	0.775442	3.28060E-05	yes	3	—	—	—
5	2	—	—	—	13	0.611090	5.41966E-05	yes
6	17	0.712305	8.28891E-05	yes	21	0.714140	1.57207E-05	yes
7	21	0.704426	1.11940E-05	yes	40	0.557180	3.76366E-13	yes
8	36	0.430888	5.21805E-14	yes	27	0.494779	6.79791E-10	yes
9	45	0.450082	0.0	yes	29	0.616717	1.42539E-08	yes

Table 6.8: Results of the Shapiro-Wilk's normality tests for the planning time variable (T_p). The table shows the Shapiro-Wilk's W statistics, p -values and the results of the tests.

We have tried to find what probability distribution the planning time belongs to. We have checked all the most usual distributions, which are available in the StatGraphics tool ². Unfortunately, we have not found any distribution that significantly fits the planning time variable. Among all the probability distributions checked, the lognormal distribution seems to be the one closest to our results. More experiments should be performed as future work to reach a proper conclusion on this subject.

²Specifically, we have employed the 16.1.07 version of the STATGRAPHICS statistics tool, named "Statgraphics Centurion XVI".

Relation between planning time and system size

In order to characterise how our approach behaves with respect to the target system size and to check its scalability, we have looked for the relation between planning time and system size. Given the results of the normality tests, we cannot apply linear regression tests, and due to the high dispersion of the data, we have relied on the median rather than the mean for comparing the different samples. We have used the Kruskal-Wallis non-parametric test [SCJ88] in order to verify, for each design smell case, the equality of medians for planning time (T_p) between the different systems. Samples with less than 5 experiments have been omitted from the tests. Nevertheless, the significance of the results does not change even if we include all the systems.

Test:	Median equality test. Kruskal-Wallis.
Variable:	T_p (Planning time)
Hypotheses:	H_0 : T_p medians are equal for all systems in case i . H_1 : T_p medians are different between some systems; ($\neg H_0$).
Cases:	case $i \in \{Feature\ Envy, Data\ Class\}$

The test results, which are displayed in Table 6.9, confirm that H_0 can be rejected with 95% confidence in all cases. Medians for T_p (planning time) are different between systems, and therefore, there is a dependency between system size (PEF) and planning time (T_p). As a consequence, it can be assured that there is some kind of relationship between the time elapsed to compute a refactoring plan and the size of the targeted system. Recalling the descriptive analysis, the data suggest the planning time increases as the target system gets bigger. Unfortunately, given the non-normality of planning time, we cannot determine the regression function. As a consequence, we cannot characterise the relationship between the two variables properly and thus, we cannot infer the planning time for a given system size. More experiments and data, at least 5 experiments per system, are needed to further investigate this in the future.

System	Feature Envy		Data Class	
	Sample size	Median	Sample size	Median
1	2	(6.4925)	7	1.5650
2	2	(8.9820)	2	(8.9275)
3	18	8.1995	16	7.2395
4	26	3.6425	3	(65.8030)
5	2	(70.8390)	12	13.5880
6	17	19.8890	21	14.8370
7	21	42.8300	40	16.4325
8	36	40.4710	27	49.4620
9	45	32.6160	29	32.5980
Test result	$K = 71.1583$; $p - value = 0$ Reject H_0		$K = 33.2451$; $p - value = 0.00000940631$ Reject H_0	

Table 6.9: Results of the Kruskal-Wallis tests for the planning time variable (T_p). The table shows the planning time medians and the results of the tests. Medians enclosed in parentheses correspond to samples with less than 5 experiments that have not been used in the tests.

Relation between planning time and case study (strategy)

We have also checked whether there are significant differences between the elapsed planning time with respect to the smell strategy requested. Unfortunately, since we have only two case studies –*Feature Envy* and *Data Class*–, this cannot be a conclusive analysis. Nevertheless, it can still serve to characterise the prototype for current and future reference. Following the same criteria as the previous test –non-normality, high dispersion– we have selected a non-parametric test and we have compared, for each system, the medians between the two design smell cases. Since, in this case, we are comparing two groups only, we have performed a Mann-Whitney (Wilcoxon) test [SCJ88]. The test has only been performed for those pairs of medians for which the size of both samples are equal or greater than 5 experiments.

Test:	Median equality test. Mann-Whitney (Wilcoxon).
Variable:	T_p (Planning time)
Hypotheses:	H_0 : T_p medians are equal between both case studies (<i>Feature Envy</i> and <i>Data Class</i>) for system i . H_1 : T_p medians are different between both case studies (<i>Feature Envy</i> and <i>Data Class</i>) for system i ; ($\neg H_0$).
Systems:	system $i \in \{1 \dots 9\}$

The results of the tests, which are shown in Table 6.10, determine that the null hypothesis cannot be rejected. Therefore, we cannot conclude, with a 95% confidence, that the planning time (T_p) is different between the smell correction strategies we have implemented. In order to determine the conclusiveness of this results, the type 2 error of the tests have been computed ([Coh88, pages 36 and 42]). Assuming normality for T_p , the test's type 2 error (β) varies between 0.69 and 0.90. According to Siegel [SCJ88], this error would be even bigger for non-parametric tests. Specifically, in the non-parametric tests we have carried out, the error should be incremented by 5% over the parametric tests' error. In our case, the test's type 2 error ranges between 0.74 and 0.95, therefore the results of these tests –do not reject H_0 – are not conclusive.

System	Feature envy		Data Class		Test result		
	Sample size	Median	Sample size	Median	W	$p - value$	Reject?
1	2	6.4925	7	1.5650	—	—	—
2	2	8.9820	2	8.9275	—	—	—
3	18	8.1995	16	7.2395	132	0.691521	no
4	26	3.6425	3	65.8030	—	—	—
5	2	70.8390	12	13.5880	—	—	—
6	17	19.8890	21	14.8370	143	0.304172	no
7	21	42.8300	40	16.4325	401	0.778845	no
8	36	40.4710	27	49.4620	469	0.818735	no
9	45	32.6160	29	32.5980	556	0.287789	no

Table 6.10: Results of the Mann-Whitney (Wilcoxon) test for checking, for each system, median equality of planning time (T_p) between case studies. The table shows the sample's medians, the Mann-Whitney's W statistics, $p - values$ and the test results.

Upper bounds for planning time

Finally, based on the information we have gathered in all the previous analyses, we have performed another attempt to give a rough estimation of how the planning time behaves with respect to system size. Although we could not determine the correlation function between planning time and system size, mean and standard deviation can still be used to define probabilistic boundaries for planning time. For this purpose, we have computed the probabilistic boundaries defined by Chebishev's inequality [SCJ88]: $Pr(|X - \mu| \geq k\sigma) \leq 1/k^2$.

Chebishev's inequality defines the probability ($1/k^2$) of a variable having an observation that is further than $k\sigma$ from the mean μ . For each system and design smell case study –*Feature Envy* and *Data Class*– we have computed the upper boundaries under which 75% and 90% of the planning times are guaranteed to be found. Boundaries are computed as $k\sigma + \mu$ and probabilities as $1 - 1/k^2$, for $k = 2$ and $k = 3.2$. The results are shown in Table 6.11 and are displayed graphically in Figure 6.17. The results in Table 6.11 should be read as “The time required for planning a *Feature Envy* strategy for a system of a similar size as the system 1 is expected to be less than 22.71 seconds in 75% of cases and less than 32.45 seconds in 90% of the cases.”

These boundaries summarise the general findings we have been able to gather with the experiments and the data we have. The mean planning times behave in a quite constrained manner even for the biggest systems. They fall below 3 minutes in all cases, and if we used medians to summarise the planning time, the results would be even better (see Table 6.6 and Figure 6.16). We cannot precisely infer how the mean planning time can evolve for systems bigger than those we have employed, due to the high dispersion of the data, but the trend seems to be promising. Despite the variability of the planning times obtained, it can still be predicted that 75% of the plan computations should fall below 15 minutes, in the case of the *Feature Envy* smells, and below 12 minutes in the *Data Class* smell cases. Broadening these boundaries, 90% of the plan computations should fall below 21 minutes for the *Feature Envy* case and 18 minutes for the *Data Class* case.

This analysis confirms that the main threat to the scalability of the approach is the high dispersion of the planning time. This issue will be one of the major concerns in future improvements and experiments. A small fraction of the requested plans can take a long time to compute. Moreover, the difference between the planning times of these extreme cases and the majority of the plan computations is quite significant. It also seems that the planning time dispersion problem is worse in the *Feature Envy* case. This can give us some hint in order to tackle the problem in future works.

As a final remark, it should also be noticed that, since Chebishev's inequality does not make any assumption about the probability distribution of the analysed variable, the given boundaries are quite poor and loose. Future experiments and additional data could help us confirm the data distribution and therefore to compute more precise and tighter boundaries.

Summary of the findings of the statistical analyses

As a final summary, the compiled results of the different statistical analyses performed regarding the objectives of the experiments are:

System	Feature Envy				Data Class			
	Mean Time	Std. Deviation	0.75	0.90	Mean Time	Std. Deviation	0.75	0.90
1	6.49	8.11	22.71	32.45	3.78	3.58	10.95	15.25
2	8.98	9.69	28.36	39.99	8.93	0.05	9.03	9.09
3	10.40	11.81	34.01	48.18	9.48	10.42	30.31	42.81
4	4.04	2.00	8.04	10.44	75.63	73.19	222.01	309.83
5	70.84	32.60	136.03	175.15	20.21	27.23	74.67	107.34
6	33.98	41.50	116.97	166.77	19.20	20.28	59.76	84.1
7	109.92	148.23	406.39	584.27	131.11	258.01	647.12	956.73
8	159.32	379.76	918.85	1374.56	143.62	297.85	739.32	1096.75
9	104.71	216.00	536.7	795.9	61.65	93.17	247.99	359.8

Table 6.11: Probabilistic upper bounds for the planning time (T_p) according to Chebishev's inequality. The 0.75 and 0.90 columns show the boundaries below which T_p is expected with the probability of the column header. Boundaries are computed as $k\sigma + \mu$ and probabilities as $1 - 1/k^2$, for $k = 2$ and $k = 3.2$.

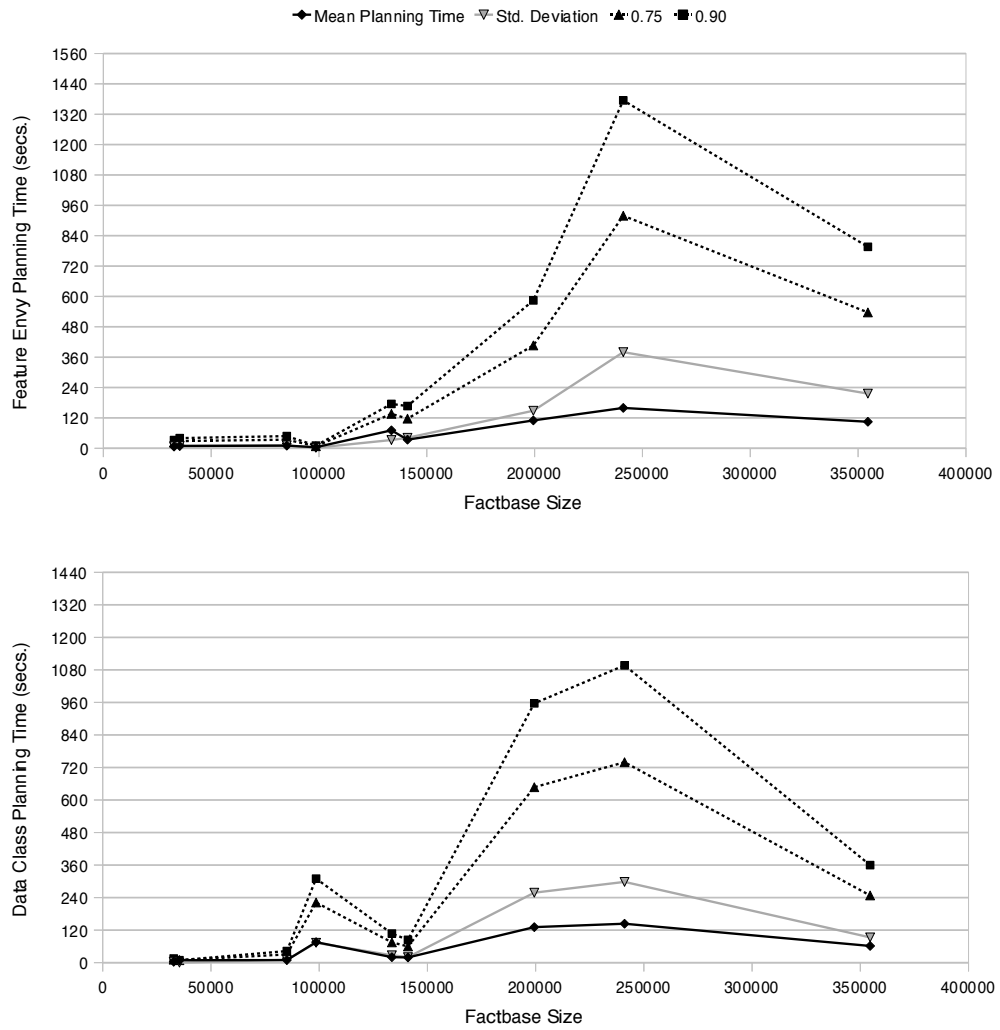


Figure 6.17: Means and upper bounds for 0.75 and 0.90 probabilistic upper bounds of planning time per factbase size, according to Chebishev's inequality. The top chart refers to the Feature Envy case, while the bottom one refers to the Data Class one.

- **Effectiveness:**

- The effectiveness of the approach, in terms of the plans produced for each case study, is quite satisfactory –42% for *Feature Envy* and 92% for *Data Class*–, specially given the relative simplicity of the refactoring planning domain implemented.

- **Efficiency and scalability:**

- Precompilation time is tightly correlated to system size.
- Planning time does not follow a normal distribution, therefore we have used non-parametric tests for all the inferential statistic analyses.
- Planning time depends on system size, although a correlation function could not be defined.
- Planning time does not depend on the strategy requested.
- The probabilistic upper bounds of the planning time for the evaluated systems are quite satisfactory, specially for a prototype.
- More experiments and data, including more systems and design smell refactoring strategies, are needed in order to obtain more conclusive results regarding planning time.

6.5 Conclusions of the case study

In this chapter we have presented a case study to evaluate the suitability of our approach to improve the automated support for complex refactoring processes. JSHOP2 is an HTN forward planner that computes the plan in the same order that it would be applied. The current state of the system is available during planning time, so this allows complex queries, calls to external procedures, etc., to be used. With this study, we have found that the efficiency of this family of planners and the expressiveness of the JSHOP2 domain specification language makes it the appropriate planner to support the refactoring planning problem. The chapter is concluded by discussing the issues we have found and the possible solutions and improvements that can be explored in the future.

Sample refactoring planning domain

The refactoring planning domain is the “core” of our approach. We have compiled and tested some refactoring strategies. Even though they are simple strategies, they illustrate how more complex strategies can be supported, and how the approach can help to improve the automation of complex refactoring processes. Due to the relative simplicity of the refactoring strategies we have compiled, simple refactoring plans are produced. The refactoring domain has been implemented for demonstration purposes and, as such, it is only able to solve a very narrow range of smells and refactoring applications. Nevertheless, this is not really an issue, since our approach has been designed to be improved by the reuse and incremental enrichment of the specifications available. The refactoring planning domain can be incrementally improved, in order to cover more design smells and refactorings, as developers compile more empirical knowledge in the form of additional refactoring strategies. Our future intentions are to increase and refine the refactoring planning domain, writing more refactorings, transformations, and refactoring strategy specifications in order to improve the refactoring plans produced.

Implementing the refactoring planning domain

One of the drawbacks of our approach is that writing HTNs for this kind of problem is quite difficult. Writing and debugging the JSHOP2 domains is a hard task. Nevertheless, once the underlying infrastructure – *i.e.* refactorings, queries, etc. – has been written and optimized, they can be easily reused. In order to hide the internal complexity of the tool, while allowing the reuse of these specifications, we have defined a domain specific language for developing new refactoring strategies. Refactoring strategies can be formulated naturally and directly from the correction recipes that a developer would write from empirically obtained knowledge. We expect the writer of refactoring strategies to use only this language. We plan to develop a compiler/translator to offer automated support for this language in the near future.

Outdated version of JTRANSFORMER

The JTRANSFORMER version we use has some limitations that affect the usefulness of our prototype. For example, it does not support JAVA generics and as a consequence, it is not able to generate the PEF factbase for systems that use generics. It cannot process and represent incomplete systems with unresolved references and library dependencies either. Luckily, there is a straightforward solution to this, because the newest version of JTRANSFORMER can parse and represent incomplete projects and systems with generics. We have not migrated our refactoring planning domain because it implies rewriting a number of queries, refactorings and transformations, but we have scheduled this as future work.

Different logic computation environments

One of the limitations of our approach is the absence of an embedded smell detection mechanism within the domain knowledge and the plan computation environment. It would be useful so that additional information could be used to guide the selection of strategy paths. Unfortunately, some of the computations needed to implement this feature, *i.e.* *metrics*, are hard to implement within the JSHOP2 planning domain, or can only be implemented in a very inefficient way. In order to solve this issue, we also plan to aim our future efforts at translating the JSHOP2 planner algorithm into PROLOG, so we can develop an integrated prototype on top of the current versions of ECLIPSE and JTRANSFORMER. This would allow us to circumvent the current limitations of the approach, for example by using the PROLOG pruning operator to implement efficient metric computations.

Impact of loosely integrated tools isolated from the IDE

Our prototype generates the system's required representation with the ECLIPSE IDE and the JTRANSFORMER plugin. This representation is processed outside the IDE in order to obtain the requested refactoring plans and a list of refactorings is produced as the output of this process. This sequence of refactorings has to be manually applied by the developer with the refactoring tools provided by the IDE. This process is acceptable for a prototype, but it is not valid for a production tool. Moreover, the prototype has been built upon a mixture of interactive and non-interactive tools. Shell scripts have been used to automatize the planning process. This kind of "integration" also has a negative impact on the efficiency of the tool. An integrated tool, directly available from the IDE, should be developed in the future for our approach to be useful in a production environment.

Efficiency issues

We have been able to extract some conclusions with respect to the efficiency of our approach in the context of our particular reference prototype. It seems that planning time increases as the target system gets bigger. However, while more experiments and data are needed to characterise this correlation, it is usually under four minutes. The approach scales for the mean planning time, but presents a high variability and thus, a small fraction of the plans can take much more time to compute. The planning time does not belong to a normal distribution. It seems to fit a lognormal distribution, but more experiments and data are needed to confirm this. If this can finally be corroborated, it would seem that it would only take a long time to compute a plan for a few rare cases, while the vast majority would be obtained within low time boundaries. We have not been able to establish correlations or good estimations to predict how long it will take a plan to compute for a system of a given size. All these issues are due to the great dispersion of the planning times, because of the big number of extreme outliers. Unfortunately, we have not found a cause for these outliers yet, so this is definitely an issue we should look at in future work.

The JSHOP2 precompilation stage adds a significant overload to the total plan computing time. In the case of this first prototype, strategy instantiations can be packed and launched within the same planning problem so the precompilation stage is performed just once. The precompilation time will become marginal as more refactoring plans are computed for the same run. Moreover, as already mentioned, we plan to develop a custom implementation of the planning algorithm with PROLOG so we can fully avoid this overload in a future version of the prototype.

As a general conclusion on efficiency, we can say that even with a prototype, comprised of loosely integrated tools, and with a demonstration implementation of a refactoring planning domain, we have obtained quite satisfactory results regarding time costs for systems of small to medium size. We expect an integrated and more mature tool could give better results in the future.

6.6 Characterisation of the refactoring planning approach

According to the taxonomy we have presented in Chapter 3, the refactoring planning approach introduced in this PhD Thesis dissertation can be described along the three main dimensions defined in the feature model (see Figure 3.2 in page 25): supported design smells, targeted artefact and supported activities.

6.6.1 Regarding design smell

Our approach can deal with any type of design smell for which a correction strategy can be written. The technical requirements we have defined for an environment supporting refactoring planning, such as the expressiveness for representing the full software system's AST, and the ability to specify and perform structural, lexical and numerical system analyses, guarantee that any kind of design smell can be managed. Nevertheless, specifically in regards of the sample prototype presented in this dissertation, the refactoring strategies developed as part of our reference implementation address two design smells: *Feature Envy* and *Data Class*. These are the only smells our prototype actually supports for now.

6.6.2 Regarding target artefact

Our approach has been specifically developed for targeting JAVA source code. Nevertheless, it can still be used for other cases. It can be applied to any scenario for which a logic-based representation of the target artefact can be obtained. Our approach can be used for other programming languages or either for different types of artefacts, such as models. In order to do that, other tools different than JTRANSFORMER [JTr] should be used for translating the target artefact into a first-order logic predicate representation and different refactoring planning domains should be written for each particular case.

As its internal representation, our approach uses first-order logic predicates that model the full software system's AST. Regarding the support for multiple system versions, our approach has not been designed to allow this, and it is not planned as future work either.

6.6.3 Regarding activities

As for the design smell activities supported, our approach is focused on specification and correction. It can be further extended to also support detection, and maybe impact analysis in a lesser extent. On the contrary, we do not see how our approach could helpfully support design smell visualisation.

Specification

Our approach supports the specification of correction strategies with domain specific languages. The JSHOP2 language [Ilg06] is preferred to define specifications of refactorings and other non-behaviour-preserving transformations. On the other hand, refactoring strategies are more easily written with the strategy specification language we have defined. As an alternative, in the case of the system queries, it could also be possible to write most of them in PROLOG and to translate them automatically into the JSHOP2 language.

The specification process is currently manual. Nevertheless, we look forward to explore how this can be automated in the future. Artificial intelligence techniques that require knowledge to be written are often improved by developing ways of automatically acquiring and refining this knowledge. In regards of the automated planning approach we use, some techniques already exists for developing and improving HTN domains automatically [CMGF⁺07, Hay08, JD10, RH07, ZHH⁺09]. Future work should explore the chances of capturing the necessary knowledge for building refactoring planning domains from the regular usage of a refactoring tool by developers.

The result of the specification activity is a refactoring planning domain definition composed of refactoring strategies, refactoring and transformation specifications and a set of system queries.

Correction

Design smell correction is the central activity supported by our approach. It is based on the specification of design smell correction recipes as refactoring strategies and the automated instantiation of them into refactoring plans for each particular case. The automation level of the approach, in its current state of development, is equivalent to "execute on approval". Our current prototype does not integrate the functionality to apply the computed plan, but the fully instantiated plan generated can be straightforwardly used to feed a refactoring tool and launch the refactoring sequence under the user request.

The result of the correction activity is a refactoring plan: A sequence of behaviour-preserving transformations that can be applied over the target artefact at its present version. As mentioned in the paragraph above, the prototype can be improved in the future in order to offer the immediate application of the generated plan. As a consequence, the result of this activity could also be the transformed artefact.

Detection

The detection of design smells is not currently designed into our approach, thus this activity is not supported. Nevertheless, reports of entity-smell relationships are introduced and used alongside the target artefact to make this information available for helping the instantiation of the refactoring strategies. Additional queries can be added to the refactoring planning domain in the future, so that the activity of design smell detection can be supported by means of rules, heuristics and metrics that might be composed into design smell detection queries.

Impact analysis

The impact analysis activity is not intended to be supported by our approach. Nevertheless, the refactoring plans generated by the refactoring planner, can be used to overview and evaluate the impact of applying the proposed design transformation. The refactoring sequence is not applied to the target artefact, but the developer has to do it. This way, the impact or the cost of applying the obtained refactoring plan can be examined at a rough level of detail in terms of the number of entities to be changed or the number of transformations in the plan.

Chapter 7

Conclusions

This chapter summarises and discusses the results of this PhD Thesis dissertation. It also reviews the differences with related approaches, details the main contributions and limitations, and presents the future work and open questions that have arisen.

7.1 General results

A historical review of the technique of improving software's structure through the detection and correction of bad smells and related design problems has been performed. The similarities and differences between the different approaches and catalogues addressing design problems have been reviewed. As a result, we have proposed a unifying terminology for these design issues, referring to them, homogeneously, as “design smells”. The current situation of the research in this field has been analysed and the automation of design smell correction activities has been identified as the next milestone.

A comprehensive survey on the subject of design smell management has been performed and a taxonomy using feature diagrams as a graphical guidance has been elaborated to illustrate it. This taxonomy can help in various ways. Newcomers to the domain can use it to become acquainted with the important aspects of design smell management. Tool builders may use it to compare and improve their tools, while software developers may use it to assess which tool or technique is most appropriate to their needs.

We have proposed a unification and generalisation of the variety of existing design smell correction approaches into the form of refactoring strategies and refactoring plans. Refactoring strategies compile the empirical knowledge on how to remove a bad smell or how to apply a refactoring when preparatory refactorings are required to enable the refactoring's precondition. Refactoring plans are instantiated from refactoring strategies and represent the precise sequence of behaviour-preserving transformations which can be immediately applied to a system at its current state. The refactoring strategies concept has been defined and elaborated as a way to allow the description and formalization of complex refactoring processes. Refactoring strategies have been defined in terms of UML class diagrams, and a simple domain specific language has also been proposed for writing them.

The main characteristics of the problem of instantiating refactoring strategies into refactoring plans have been stated. These characteristics define which approaches are suitable for addressing the instantiation of refactoring plans. Thus, they have been used to filter out and select the most appropriate approach. The instantiation of refactoring strategies has been represented as

an automated planning problem. We have selected HTN planning and, particularly, JSHOP2 as the most appropriate planning approach and planner available for the instantiation of refactoring strategies into refactoring plans.

We have stated that refactoring plans can be computed with an automated planner, specifically, by implementing refactoring strategies as task networks for a HTN planner. In order to demonstrate this, we have defined how refactoring strategies can be specified as JSHOP2 task networks and how the refactoring planning problem can be addressed as a JSHOP2 planning problem. The elements of refactoring strategies have been formulated as JSHOP2's HTN elements. We have also formulated the instantiation of refactoring strategies into refactoring plans as a JSHOP2 planning problem.

A sample prototype has been assembled to demonstrate our approach. Some existing tools from other authors have been integrated for building the basic prototype's infrastructure and a refactoring planning domain has been developed: a HTN domain specification comprising system queries, refactoring strategies, specifications of refactorings and other non-behaviour-preserving transformations.

Two case studies have been carried out to evaluate our approach, and the reference prototype has been tested in terms of effectiveness, efficiency and scalability. The case studies used are addressed for removing the *Feature Envy* and *Data Class* design smells and have been performed over 9 software systems of different sizes ranging from small to medium size. The results of the study confirm that our approach can be used to automatically generate refactoring plans for complex refactoring processes in a reasonable time. The studies performed also demonstrate that the efficiency of the HTN family of planners and the expressiveness of the JSHOP2 domain specification language makes it the appropriate planner to support the refactoring planning problem.

7.2 Results regarding thesis statements

For reference purposes, the main Thesis statement, formulated in Chapter 1, is reproduced again here:

The activity of refactoring, when complex refactoring sequences have to be applied, as in the case of design smell correction in Object-Oriented software, can be assisted by means of refactoring plans that can be obtained automatically.

The discussion on how the main Thesis statement has been addressed is performed by reviewing the thesis statements in which the main one was decomposed. The thesis statements listed in Chapter 1 are recalled here and discussed individually:

- *State of the art in automated design smell management is mature in detection but still has to be improved in correction.*

The revision of the state of the art performed in Chapters 2 and 3 has served to demonstrate this. The degree of automation achieved by the design smell detection tools and approaches has reached the “fully automated” level in many cases. Moreover, the results, as claimed by the authors, are quite acceptable and, as already mentioned, there is a chance of seeing industry tools based on them soon.

Nevertheless, the correction of design smell has barely materialised into tools. There are few approaches dealing with design smell correction, the catalogues of correction “recipes” being the main reference of the current state of the art in this design smell management activity. The “manual” level of automation is, therefore, the maturity level achieved in general. To the best of our knowledge, there have only been a few heuristic implementations of correction procedures for very specific smells. The “fully-automated” level of automation has been achieved for these few particular and individual cases. However, in general, as already mentioned in Chapter 2, design smell correction lacks more successful work on precise and systematic specification and automation.

- *Refactoring Suggestions produced by current design smell detection tools are not directly applicable.*

The revision of the state of the art performed in Chapters 2 and 3 has also served to demonstrate this. Those approaches dealing with design smell correction are only able to produce suggestions for correction. These suggestions are based on general strategies and are produced without taking into account the current state of the system or how the suggestion should be applied for the particular case. An exception is provided by the few approaches and tools focused on single smells, developed for applying specific strategies for particular smells.

- *Design smell correction with refactorings corresponds to the general schema of applying complex refactoring sequences with a strategic objective.*

Chapter 4 has been dedicated to analysing how design smell corrections have been specified in the literature until now. How complex refactorings are specified has also been reviewed. We have also analysed how both types of transformations are applied. This Thesis statement has therefore been addressed by elaborating the refactoring strategy concept which represents design smell correction strategies in particular, and specifications of complex refactoring processes in general. Smell correction strategies and refactoring mechanics can be described homogeneously as complex refactoring processes. As a consequence, both can be defined as refactoring strategies and refactoring plans can be instantiated from them.

- *Complex refactorings can be assisted with refactoring planning, by enabling refactoring preconditions with preparatory refactorings that can be obtained automatically.*

To address this Thesis statement, we have developed an approach and a prototype, based on HTN planning, which can be used to instantiate refactoring strategies into refactoring plans. A sample refactoring planning domain has been written to be used as the domain knowledge for a planner. This domain contains the specification of a complex refactoring –**MOVE METHOD**– along with a strategy for computing the necessary preparatory refactorings in order to increase the applicability of this refactoring. The approach has then been evaluated with case studies involving the use of the **MOVE METHOD** refactoring.

- *Design smell correction, as a special kind of complex refactoring process, can be assisted by means of refactoring planning.*

This Thesis statement addresses a more general case than the previous one and it is aimed at a more complex objective. Refactoring strategies for two design smells –*Feature Envy* and *Data Class*– have been written and tested experimentally, and successful results regarding effectiveness, efficiency and scalability have been obtained.

7.3 Results regarding thesis objectives

The Thesis objectives formulated in Chapter 1 are reproduced here and discussed individually:

1. *Provide an automated or semi-automated support to plan ahead the preparatory refactoring sequences –refactoring plans– that can enable the precondition of a desired set of refactorings.*

An approach has been elaborated based on refactoring strategies that are instantiated into refactoring plans by means of HTN planning. The approach allows the empirical knowledge gathered by software developers, refactoring and reengineering practitioners to be formalised into “recipes” about how to apply a particular refactoring, we have named “refactoring strategies”.

The approach allows the exact refactoring sequence that needs to be applied for each particular case, to be computed from the refactoring strategies. We have named these sequences “refactoring plans”. The specific refactoring plan obtained for each case contains all the necessary preparatory refactorings that have to be executed to allow the application of the particular desired refactoring. This has been demonstrated by writing a refactoring strategy for assisting the application of the **MOVE METHOD** refactoring.

The approach has been designed to be fully automated, but user interaction can be requested for cases in which the inference mechanism is not smart enough and additional information has to be collected from the user during the refactoring plan computation process.

2. *Provide an automated or semi-automated support to assist the generation of refactoring sequences –refactoring plans– that can transform a system, following a redesign proposal while preserving the system’s behaviour. More specifically, to provide an automated or semi-automated support to the generation of refactoring plans for design smell correction.*

Refactoring strategies and refactoring plans have been defined in such a way that our approach can be used for design smell correction. The refactoring planning approach we have developed can address the purpose mentioned in objective 1 but, in a more general way, it can also be used for computing the refactoring plan for any complex refactoring process aimed at a particular objective.

Specifically, we have demonstrated that our approach can be applied to the problem that motivates this PhD Thesis dissertation: the correction of design smells. Refactoring strategies for removing *Feature Envy* and *Data Class* design smells have been written and tested for this purpose.

3. *Provide a way to help software developers use the techniques elaborated in this PhD Thesis Dissertation.*

Our refactoring planning approach has been designed in such a way that it allows three different kinds of software developers to use it.

The most common users of the approach would simply use it to find a way to apply a desired complex refactoring process. They would select the strategy they want to apply, fill in the required parameters, launch the strategy instantiation and wait for the planner to produce a refactoring plan. The refactoring plan could be then applied with the help of a refactoring tool.

A developer with enough experience in refactoring and reengineering would also be able to write customised refactoring strategies. This is allowed by the strategy specification language proposed in this Thesis and the system queries that hide the complexity of the internal representation of the system's AST. Moreover, refactoring strategies can be shared and reused. Therefore, these developers can contribute to, or take advantage of, public shared catalogues of refactoring strategies.

A developer with a deeper knowledge of our approach, such as the logics-based internal representation employed, and specially with HTN planning, would be able to write system queries, non-behaviour transformations and refactoring specifications.

4. *Evaluate the effectiveness, efficiency and scalability of the approach presented in this PhD Thesis Dissertation by developing a prototype which implements this approach and by performing an experimental study with it.*

A reference prototype has been assembled that implements our refactoring planning approach and serves to demonstrate it. An experimental evaluation of the approach has then been performed by carrying out two case studies dedicated to removing *Feature Envy* and *Data Class* design smells over nine different systems, ranging from small to medium size.

7.4 Characterisation of the approach

According to the taxonomy presented in Chapter 3, our approach can be described along the three main dimensions defined in the feature model (see Figure 3.2): supported design smells, targeted artefacts and supported activities.

Regarding the **Design Smells** addressed, our approach can deal with any type of design smell for which a correction strategy can be written. The requirements we have established for our approach and the internal representation used guarantee that any kind of design smell can be managed. Nevertheless, the reference prototype presented in this dissertation only supports two design smells: *Feature Envy* and *Data Class*.

Regarding the **Target Artefact**, our approach deals with JAVA source code, uses first-order logic predicates, which model the full software system's AST, as its internal artefact representation and does not support multiple system versions. The approach can also be adapted and modified to support other programming languages and artefacts. It can indeed support any target artefact for which a logic-based representation can be obtained.

Regarding the **Activities** supported, our approach can address: *specification* in a manual way, for which the activity result is a refactoring planning domain definition; and *correction* at an "execute on approval" automation level, for which the activity result is a refactoring plan. The rest of the design smell management activities we identified in the taxonomy are not supported: *detection* is not currently supported, but it may be in the future, *impact analysis* could also be offered to some degree but *visualisation* is not supported and neither is this planned in the future.

7.5 Comparison with related works

The only similar works we have knowledge of, regarding design smell correction, are those from Trifu *et al.* [TSG04, Tri08]. They present an approach to define and apply restructuring strategies

that specify how to remove a design smell in a similar recipe-based way as our refactoring strategies. Our refactoring strategies are indeed inspired by the concept of restructuring strategies from these authors. However, they lack the instantiation of the strategies that our approach provides, supported by automated planning.

Our approach applies automated planning, an artificial intelligence technique, in order to solve a software engineering problem, computing the refactoring sequences needed for performing complex refactoring processes. To the best of our knowledge, this is a novel approach, because we have not found any other work that uses automated planning for this particular purpose. Moreover, we have found only a few references on the usage of automated planning in software engineering.

Memon *et al.* have used automated planning to develop a tool for generating GUI test cases [MPS01]. In their work, the user interface is analysed to obtain a list of the allowed events as the applicable operators. The test designer would then define the preconditions and effects of the different events the user can trigger and the initial and goal states of the testing scenarios are also specified. The plans produced by the planner would represent testing sequences aimed at covering the whole set of the GUI's possible states. They use the Interference Progression Planner (IPP) [KNHD97], a partial order planner that belongs to the GRAPHPLAN family of planners. Their choice of planner seems to be appropriate for the problem they tackle. Nevertheless, their results cannot be compared against ours since the two problems differ greatly in nature and, probably, in size. Moreover, their experiments were performed in 2001 with a PENTIUM®-based computer. According to the authors, their GUI testing domain is composed of 32 planning operators but, unfortunately, they do not specify the size of the system's state. We can nonetheless guess that it cannot be even close to the 32780 to 354543 predicates in the system states from our problem. Despite the fact that their results are not comparable with ours, their experiments also take into account hierarchical planning techniques, which are related to HTN planning. The experiments performed by the authors with and without hierarchical planning techniques reveal the relevance of HTN planning with respect to efficiency.

HTN planning, the SHOP2 planner in particular, has been used for web service composition by Sirin *et al.* [SPW⁺04]. They address the problem of selecting which web services have to be invoked, from among a set of available services, and in which order, to build up a more complex composed web service. Although their problem is different in nature and size, some of the techniques we have used for materialising our approach are similar to theirs. In their work, the HTN domain definition is obtained from the available web services specification –written in the Web Ontology Language (OWL), reference version 1.0 [DCvH⁺02]– by translating these specifications into HTN elements. This translation is defined with systematic rules. A similar approach has been followed here for translating refactoring strategies, and the refactoring strategy language, into HTN elements. We have even translated some elements, such as loops, in a similar way to them. We have also used persistent variables for similar purposes as they do in their work. They have also used, as we do, external procedure calls from the planner in order to query the user for additional information during the planning process.

Pinna *et al.* have explored the suitability of partial order planning for dealing with model inconsistencies [PVDSM10]. Their system states represent UML models, particularly class diagrams and consistency rules. Their planning operators represent simple model transformations. They search for plans aimed at removing model inconsistencies, represented as violations of the consistency rules. They have experimented with forward and backward partial order planners, and found that an approach based on these kinds of planners presents severe efficiency issues even

for very small problems. As they have concluded, a planner that searches the whole state-space without any domain-knowledge guidance does not scale well for big problems. This is consistent with our findings, which were described in Chapter 5.

7.6 Limitations of the approach

The requirement of writing an HTN domain –the refactoring planning domain– is the main limitation of the approach presented in this PhD Thesis dissertation. Our approach does not compute refactoring plans by exploring the whole search space, the planner instantiates them from heuristics that have already been defined in the form of refactoring strategies. These specifications have to be gathered from empirical knowledge and translated into HTNs. As a consequence, the effectiveness of the approach depends greatly on the amount of refactoring strategies, refactoring specifications, transformations and system queries defined and implemented in the refactoring planning domain. How refactoring strategies’ specifications can be found has not been addressed here. Writing a big amount of comprehensive strategies is beyond the scope of this dissertation. Rather, we have focused on defining the approach and building a reference implementation in order to evaluate and demonstrate it. Therefore, the number of design smells, and the particular manifestations of them, that our prototype can manage are somewhat limited. Nevertheless, the refactoring planning domain we have written serves as a reference and can be enlarged and improved in the future.

As another drawback, writing knowledge for the refactoring planning domain is hard, specially without any tool or assistance, as we have done during the elaboration of this dissertation. It is a complex and error-prone task if it is manually performed. Nevertheless, this limitation can be avoided by developing custom tools to assist the development and maintenance of the refactoring planning domain. As an initial step, a domain specific language has been proposed for writing refactoring strategies and we have discussed in Chapter 5 how the different elements of a refactoring planning domain can be translated into JSHOP2 HTNs.

Some computations are hard to implement in PROLOG or the JSHOP2 language. Our approach can only use the structural, lexical and numerical information that can be obtained from the target system’s AST. Some information is difficult to infer. For example, in order to write a refactoring strategy for applying the **EXTRACT METHOD** refactoring, the planner should be able to find the portions of the method to extract. There may also exist problems when computing plans that need semantic information for example, if a refactoring strategy needs to identify whether a class is a GUI class or a library class. These issues have not been addressed here. Nevertheless, for these cases, our approach can still obtain the needed information by querying the user. If the needed information, which cannot be computed through the inference mechanisms of the planner, can still be computed in some way by means of some other tools, these can be “plugged” into the planner as external procedures.

The tool developed during the elaboration of this PhD Thesis dissertation is a prototype that only serves the purpose of evaluating and demonstrating the approach presented here. Although we have been able to demonstrate the feasibility of our approach, further work is still needed in order to materialize these results into a tool that could actually be used. As listed in the results of the prototype evaluation in Chapter 6, our approach presents some applicability restrictions due to technical limitations in the reference prototype. For example, our prototype cannot deal with JAVA systems that use generics. These limitations can easily be overcome, in order to produce a better integrated and actually usable tool, either by updating the prototype for using the most

recent versions of the tools used here, such as JTRANSFORMER, or by reworking the prototype.

Due to the great dispersion in the results of the experiments regarding planning time –the elapsed time for computing the refactoring plans–, we have not been able to give a prediction or estimation on how long it would take for the prototype to obtain a plan depending on the target system’s size. Nevertheless, in our opinion, for the systems we have used in the experiments, the results have been rather good.

7.7 Summary of contributions

This PhD Thesis dissertation presents a novel approach on the automation of complex refactoring processes, and in particular on the correction of design smells. It also presents a novel application of automated planning. We do not know of other approaches employing automated planning for this particular problem –computation of refactoring plans for removing design smells. Moreover, as already discussed in Section 7.5, automated planning, to the best of our knowledge, has scarcely been used in software engineering.

This approach constitutes an appropriate infrastructure to automate design smell correction strategies, because it allows planning ahead for the precise and applicable transformation sequence for each particular case. Refactoring strategies are written in the same homogeneous way for the application of complex refactorings, for the correction of design smells, or for other similar objectives that can be explored in the future. Due to the use of an automated planner, non-deterministic constructs are available, so it allows specifications of complex refactoring processes to be written as refactoring strategies. These specifications can be very expressive, and can be directly used to automate the heuristic rules that would be written with natural language. The approach allows refactoring strategies to be specified in a modular way, and to be reused and shared.

The contributions of this PhD Thesis Dissertation are listed here. This can overlap in some way with the Thesis results previously enumerated in Section 7.1, but it emphasises that these results are contributions to the state of the art. The contributions of this PhD Thesis dissertation can be summarised as:

- A review of the design smells’ literature, a historic overview of design smell management approaches, and a terminology proposal, aimed at clarifying and unifying the terms and concepts related to this subject.
- A survey on design smell management and a taxonomy, based on feature models and co-written with other authors, to characterise the present and future approaches and tools.
- The definition of refactoring strategies as a way of writing automation-suitable specifications of complex refactoring processes.
- The definition of a refactoring strategy specification language that software developers can use to write strategies for design smell correction and other complex refactoring processes.
- The definition of refactoring plans as specific refactoring sequences, instantiated from refactoring strategies, that can be effectively applied to a system in its current state.
- The definition of the requirements that an approach has to fulfill in order to support the computation of refactoring plans.

- A technique to instantiate refactoring strategies into refactoring plans by means of automated planning.
- A base line and reference prototype for future research in automated refactoring planning.

7.8 Future work and open questions

Our first priority in the near future is to address the current limitations of our approach. In particular, we hope to improve the refactoring planning domain and to evolve the reference prototype into an actually usable tool.

In order to improve and enlarge the refactoring planning domain, firstly, it has to be adapted to use the latest version of JTRANSFORMER's logics-based AST representation. Then, the amount of cases covered by the refactoring planning domain has to be increased by adding more refactoring strategies for design smell correction, more refactoring specifications and more strategies for the application of complex refactorings. The refactoring planning domain also has to be improved by adding more system queries, specially for computing metrics. This kind of queries would provide the opportunity to implement the detection strategies from Lanza and Marinescu [LM06] into our approach as system queries. This could greatly improve the plans produced by allowing the planner to select the transformations more precisely, thus addressing the elements causing a design smell.

We should also address in the future how some complex computations could be performed or “plugged” into our approach. For example, as already mentioned, how semantic information can be obtained, how to infer the portions of a method to be extracted during a **EXTRACT METHOD** refactoring, how to distinguish a GUI class from a library class, etc., should all be explored.

As already mentioned in the conclusions of the case study (Chapter 6), we intend to rework the reference prototype and evolve it into an actually usable tool. A simple prototype has been developed using existing technology and other prototype tools. Further development of a more optimal implementation has to be undertaken. It will be based on a close connection to the latest version of JTRANSFORMER and will involve writing the JSHOP2 algorithm into PROLOG, and building a more integrated tool as an ECLIPSE plugin.

Alternative internal representations can also be used in the future for increasing the applicability of the approach. For example, an Object-Oriented software representation model such as MOON [Cre00, LMCP06, MLCP07] could be explored. This model provides support for language independence and generics, and therefore, can increase the applicability of our approach.

We also consider that it is necessary to develop tools for assisting in the writing of the HTN planning domain. A compiler for the refactoring strategy specification language has to be developed. We would also like to offer a tool to help in the writing and debugging of refactoring strategies and refactoring specifications in a visual way, with a graphical editor. A tool like this could be based on model transformations, so visually developed refactoring specifications can be automatically transformed into HTNs by following the rules elaborated in Chapter 5.

It would be helpful to develop a tool for running batteries of experiments, and for easily producing an analysis of the results. This testing tool would be used for developing the refactoring planning domain and it could help to evaluate the domain in more detail as well. For example, it will be interesting to find out the most adequate amount of knowledge needed to obtain the best results regarding effectiveness and efficiency.

Our prototype only returns the first plan found. As another way of improving the tool, we would like to explore the possibility of dealing with multiple plans. It should be useful to be able to obtain multiple plans, measure their differences in terms of quality factors and, either offer the user the best plan, or inform about the different possibilities and their respective quality.

Another line of future work could be dedicated to exploring other possible uses of our approach, different from the ones which have motivated our study and that have been presented in this PhD Thesis dissertation. As an example, it would be interesting to write refactoring strategies for other objectives such as introducing design patterns. Finally, we would like to explore how to extend the capabilities of our approach by exploring alternative techniques to support it or by testing it in completely different usage scenarios.

A different usage scenario for the approach could be the research and development of “regular” refactoring implementations for IDEs or refactoring tools. Mature refactoring strategies could be transferred to a refactoring engine as high-level refactorings. Our approach could be used to incrementally develop a refactoring strategy up to a level of detail in which the set of knowledge defined is so exhaustive and comprehensive that the inference mechanisms and non-determinism of the planner are not needed. A refactoring strategy with this level of detail could be taken from the planner and be implemented as regular refactorings within an IDE or other kind of refactoring tool.

References

- [ADN05] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 62–71, 2005.
- [agg] AGG Graph transformation tool, Graph Grammar Group, Technische Universität Berlin. <http://tfs.cs.tu-berlin.de/agg/index.html>.
- [AKN09] Ron Alford, Ugur Kuter, and Dana Nau. Translating htms to pddl: A small amount of domain knowledge can go a long way. In *International Joint Conference on Artificial Intelligence (IJCAI)*, July 2009.
- [AL08] László Angyal and László Lengyel. Refactoring System - VMTS. In *4th International Workshop on Graph-Based Tools: The Contest (GraBaTs 2008)*, September 2008. <http://www.fots.ua.ac.be/events/grabats2008>.
- [Ana08] Analyst4j. <http://www.codeswat.com>, February 2008.
- [Apa] Apache Lucene Java. The Apache Software foundation, <http://lucene.apache.org>.
- [Aqr02] Aqris. RefactorIT. <http://www.refactorit.com>, 2002. g.
- [Arg] ArgoUML. <http://argouml.tigris.org>. Tigris.org g.
- [Arn89] Robert S. Arnold. Software Restructuring. *Proceedings of the IEEE*, 77(4):607–617, Apr 1989.
- [AS09] El Hachemi Alikacem and Houari A. Sahraoui. A metric extraction framework based on a high-level description language. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 159–167, Washington, DC, USA, 2009. IEEE Computer Society.
- [BAMN06] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A novel approach to optimize clone refactoring activity. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1885–1892, New York, NY, USA, 2006. ACM.
- [BCT07] Thierry Bodhuin, Gerardo Canfora, and Luigi Troiano. Sormasa: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. In *1st Workshop on Refactoring Tools (WRT'07)* [DC07], pages 23–24.

- [BEG⁺06] P. Baker, D. Evans, J. Grabowski, H. Neukirchen, and B. Zeiss. TRex - the refactoring and metrics tool for TTCN-3 test specifications. pages 90–94, Aug. 2006.
- [BF97] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.
- [BF99a] Kent Beck and Martin Fowler. *Bad Smells in Code*, chapter 3. In *Refactoring: Improving the Design of Existing Code* [FBB⁺99], 1 edition, June 1999.
- [BF99b] Kent Beck and Martin Fowler. *Big Refactorings*, chapter 12. In *Refactoring: Improving the Design of Existing Code* [FBB⁺99], 1 edition, June 1999.
- [BGQR07] Giulia Bruno, Paolo Garza, Elisa Quintarelli, and Rosalba Rossato. Anomaly Detection in XML Databases by means of Association Rules. In *DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications*, pages 387–391, Washington, DC, USA, 2007. IEEE Computer Society.
- [BMMIM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, March 1998.
- [BMST99] Roswitha Bardohl, Mark Minas, Andreas Schürr, and Gabriele Taentzer. *Application of Graph Transformation to Visual Languages*, chapter 1, pages 105–180. Volume 2 of Ehrig et al. [EKR99], 1999.
- [Bor] Borland. Together. <http://www.borland.com/us/products/together/index.html>.
- [Bro75] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, Reading, MA , USA, 1975.
- [BTS00] Paolo Bottoni, Gabriele Taentzer, and Andy Schürr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 59, Washington, DC, USA, 2000. IEEE Computer Society.
- [CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Feature Modeling*, chapter 5, pages 83–116. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, June 2000.
- [CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
- [Che04] CheckStyle. <http://checkstyle.sourceforge.net>, 2004.
- [Chi02] Ciprian-Bogdan Chirila. Automation of the design flaw detection process in object-oriented systems. *International Conference on Technical Informatics CONTI 2002; Periodica Politecnica – Transactions on Automatic Control and Computer Science*, 47, October 2002.

- [Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 18–32, Washington, DC, USA, 1999. IEEE Computer Society.
- [CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *ACM SIGPLAN Notices*, 26(11):197–211, 1991.
- [CLMM06] Yania Crespo, Carlos López, Esperanza Manso, and Raúl Marticorena. *From bad smells to refactoring, metrics smoothing the way*, chapter VII, pages 193–249. Object-Oriented Design Knowledge. Principles, Heuristics and Best Practices. Idea Group Publishing, 2006.
- [CMGF⁺07] Luis Castillo, Lluvia Morales, Arturo González-Ferrer, Juan Fernández-Olivares, and Óscar García-Pérez. Current topics in artificial intelligence. chapter Knowledge Engineering and Planning for the Automated Synthesis of Customized Learning Designs, pages 40–49. Springer-Verlag, Berlin, Heidelberg, 2007.
- [CNYM99] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*, volume 5 of *International Series in Software Engineering*. Springer, October 1999.
- [Coh88] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum associates Publishers, 2nd edition edition, 1988.
- [Con99] Clarkware Consulting. JDEPEND. <http://clarkware.com/software/JDepend.html>, 1999.
- [Cre00] Yania Crespo. *Incremento del potencial de reutilización del software mediante refactorizaciones*. PhD thesis, Universidad de Valladolid, 2000.
- [Cul] Cultivate. ROOTS Research Group; <http://sewiki.iai.uni-bonn.de/research/cultivate/start>.
- [dbX] dbXML. Tom Bradford, Burak Bari, <http://sourceforge.net/projects/dbxml-core>.
- [DC07] Danny Dig and Michael Cebulla. 1st workshop on refactoring tools (wrt'07). Technical Report 2007-8, TU Berlin, July 2007.
- [DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automatic detection of refactorings in evolving components. In *ECOOOP 2006 - Object-Oriented Programming; 20th European Conference, Nantes, France, July 2006, Proceedings*, pages 404–428, 2006.
- [DCvH⁺02] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Web ontology language (owl) reference version 1.0. W3C Working Draft, November 2002.

- [DDGM07] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC/FSE 2007: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, USA, September 2007. ACM Press.
- [DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–177, New York, NY, USA, 2000. ACM Press.
- [DDN08] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.
- [DDV⁺06] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering*, pages 346–355. IASTED/ACTA Press, 2006.
- [DLOV08] Andrea De Lucia, Rocco Oliveto, and Luigi Vorraro. Using structural and semantic metrics to improve class cohesion. *ICSM 2008. IEEE International Conference on Software Maintenance*, pages 27–36, October 2008.
- [DMJN08] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, 2008.
- [DSA⁺04] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, July 2004.
- [DSRS03] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139, 2003.
- [Dur07] Nakul Durve. Coderaider: Automatically improving the design of code. diploma thesis, June 2007.
- [Ecl] Eclipse Metrics Plugin. <http://eclipse-metrics.sourceforge.net>.
- [ED00] L. Etzkorn and H. Delugach. Towards a semantic metrics suite for object-oriented design. In *Technology of Object-Oriented Languages and Systems, 2000. TOOLS 34. Proceedings. 34th International Conference on*, pages 71–80, 2000.
- [EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume II: Applications, Languages and Tools*, volume 2. World Scientific, 1999.
- [EGK⁺01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

- [EHN94] K. Erol, J. Hendler, and D. S. Nau. Htn planning: Complexity and expressivity. In *National Conference on Artificial Intelligence (AAAI)*, 1994.
- [EJ05] Niels Van Eetvelde and Dirk Janssens. Refactorings as graph transformations. Technical report, Universiteit Antwerpen, 2005.
- [EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume III: Concurrency, Parallelism, and Distribution*, volume 3. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.
- [EM02] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In Arie Van Deursen and Elizabeth Burd, editors, *Proceedings of the 9th Working Conference on Reverse Engineering*, pages 97–107. IEEE Computer Society, October 2002.
- [ERT99] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. *The AGG approach: language and environment*, chapter 14, pages 551–603. Volume 2 of Ehrig et al. [EKR99], 1999.
- [FBB⁺99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1 edition, June 1999.
- [FN71] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [FN99] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.
- [FTC07] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of feature envy bad smells. *ICSM 2007. IEEE International Conference on Software Maintenance*, pages 519–520, Oct. 2007.
- [FXC06] FXCop. [http://msdn.microsoft.com/en-us/library/bb429476\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx), June 2006.
- [GA08] Yann-Gaël Guéhéneuc and Giuliano Antoniol. DeMIMA: A Multi-layered Framework for Design Pattern Identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, September/October 2008.
- [GAA01] Yann-Gaël Gueheneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. *TOOLS 39. 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, pages 296–305, 2001.
- [Gan] GanttProject. GanttProject Team, <http://www.ganttproject.biz/>.
- [GDMR04] Tudor Girba, Stephane Ducasse, Radu Marinescu, and Daniel Ratiu. Identifying entities that change together. In *Proceedings of 9th IEEE Workshop on Empirical Studies of Software Maintenance (WESS 2004)*, Chicago, 2004.

- [Gei08] Leif Geiger. Graph transformation-based Refactorings using Fujaba. In *4th International Workshop on Graph-Based Tools: The Contest (GraBaTs 2008)*, September 2008. <http://www.fots.ua.ac.be/events/grabats2008>.
- [GFGP06] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Fundamental Approaches to Software Engineering (FASE)*, number 3922 in Lecture Notes in Computer Science, pages 411–425. Springer, 2006.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.
- [GKB07] T. Genssler, V. Kuttruff, and F. Brosig. Inject/J Software Transformation Language – Language Specification. Reference Manual, November 2007.
- [GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, 1993.
- [GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning; Theory and Practice*. Morgan Kaufmann, 2004.
- [Gro] Groom. Mathieu Allory, <http://sourceforge.net/projects/groom> g.
- [Gué03] Yann-Gaël Guéhéneuc. *A framework for design motif traceability*. PhD thesis, École des Mines de Nantes; University of Nantes, July 2003.
- [GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin/Heidelberg, 1999.
- [GZ10] Leif Geiger and Albert Zündorf. Fujaba case studies for GraBaTs 2008: lessons learned. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:287–304, 2010. 10.1007/s10009-010-0152-z.
- [Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.
- [Ham07] Hammurapi. <http://www.hammurapi.biz>, October 2007.
- [Hay08] Andrew William Hay. Automatically extending the coverage of hierarchical task network planners. Master’s thesis, University of Auckland, 2008.
- [icaa] icaps. Main site of the 2002 international planning competition. <http://planning.cis.strath.ac.uk/competition> g.
- [icab] icaps. Main site of the international planning competition. <http://ipc.icaps-conference.org/> g.
- [Ilg06] Okhtay Ilghami. Documentation for JSHOP2. Technical Report CS-TR-4694, Department of Computer Science, University of Maryland, College Park, MD 20742, USA, May 2006.

- [IN03] Okhtay Ilghami and Dana Nau. A general approach to synthesize problem-specific planners. Technical Report CS-TR-4597, UMIACS-TR-2004-40, University of Maryland, October 2003.
- [Int08] Intooitus. Incode infusion. <http://www.intooitus.com/>, 2008. g.
- [iPl] iPlasma. LOOSE Research Group;.
- [ISO01] ISO and IEC. ISO/IEC 9126-1:2001, software engineering – product quality, part 1: Quality model, 2001.
- [JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen.NET, the expressive, convenient and fast graph rewrite system. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:263–271, 2010. 10.1007/s10009-010-0148-8.
- [JD10] Li Jin and Keith S. Decker. Ontology oriented exploration of an htn planning domain through hypotheses and diagnostic execution, 2010.
- [JFr] JFreeChart. Object Refinery Limited, <http://www.jfree.org/jfreechart>.
- [JFRS07] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the Detection of Snapshot Isolation Anomalies. In *VLDB '07: Proceedings of the 33rd International Conference on Very Large Databases*, pages 1263–1274. VLDB Endowment, 2007.
- [JPs] Java PEF specification. ROOTS group, http://sewiki.iai.uni-bonn.de/research/jtransformer/api/java/pefs/2.9/java_pef_overview.
- [JSHa] JSHOP2. Department of Computer Science, University of Maryland, <http://www.cs.umd.edu/projects/shop>.
- [JSHb] JSHOP2. JSHOP2 download site. <http://sourceforge.net/projects/shop/files/>.
- [JTo] JTombstone. Bill Alexander, <http://jtombstone.sourceforge.net>.
- [JTr] JTransformer Engine. ROOTS group, <http://roots.iai.uni-bonn.de/research/jtransformer/>.
- [Jwe] Jwebap. <http://sourceforge.net/projects/jwebap>.
- [Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Signature Series. Addison-Wesley Professional, August 2004.
- [KG06] Cory Kapser and Michael W. Godfrey. “Cloning Considered Harmful” Considered Harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.
- [KIAF02] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proc. International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2002.

- [KK03] Günter Kniesel and Helge Koch. Program-independent composition of conditional transformations. Technical Report IAI-TR-03-1, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, July 2003. updated Feb. 2004.
- [KK04] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004. Special issue on Program Transformation, edited by Ralf Lämmel, ISSN: 0167-6423, digital object identifier <http://dx.doi.org/10.1016/j.scico.2004.03.002>.
- [KNHD97] Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an adl subset. In Sam Steel and Rachid Alami, editors, *Recent Advances in AI Planning*, volume 1348 of *Lecture Notes in Computer Science*, pages 273–285. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63912-8_92.
- [Kni06] Günter Kniesel. A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006.
- [KPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *16th Working Conference on Reverse Engineering (WCRE)*, pages 75–84. IEEE Computer Society, 2009.
- [KRW07] Douglas Kirk, Marc Roper, and Murray Wood. A heuristic-based approach to code-smell detection. In *1st Workshop on Refactoring Tools (WRT’07)* [DC07], pages 55–56.
- [KVGS09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari A. Sahraoui. A bayesian approach for the detection of code and design smells. In *International Conference on Quality Software*, pages 305–314. IEEE Computer Society, 2009.
- [Leh80] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.
- [Leh96] Meir M. Lehman. Laws of software evolution revisited. In *EWSPT ’96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.
- [LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.
- [LMCP06] Carlos López, Raul Marticorena, Yania Crespo, and Javier Pérez. Towards a language independent refactoring framework. In *1st ICSOFT 06 International Conference on Software and Data Technologies. Setubal, Portugal. ISBN: 972-8865-69-4*, volume 1, pages 165–170, sep 2006.

- [LN07] Marián Lekavý and Pavol Návrat. Expressivity of STRIPS-like and HTN-like planning. In *1st KES International Symposium*, volume 4496 of *Lecture Notes in Artificial Intelligence*, pages 121–130. Springer-Verlag, 2007.
- [LS07] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, July 2007.
- [LWN07a] Ángela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Assessing the impact of bad smells using historical information. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 31–34, New York, NY, USA, 2007. ACM.
- [LWN07b] Ángela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In *Proceedings of the 4rd International Workshop on Mining Software Repositories: IEEE Computer Society*, May 2007.
- [MA06] Beatriz Martín Arranz. Conversor de Java a grafos AGG para Eclipse. Final degree project, September 2006. Escuela Técnica Superior de Ingeniería Informática; Universidad de Valladolid.
- [Män03] Mika Mäntylä. Bad smells in software - a taxonomy and an empirical study. Master's thesis, Helsinki University of Technology, May 2003.
- [MAP⁺08] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2007), Revised Selected Papers*, pages 252–266, Berlin, Heidelberg, 2008. Springer-Verlag.
- [Mar01] Radu Marinescu. Detecting design flaws via metrics in Object-Oriented systems. In *TOOLS '01: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, page 173, Washington, DC, USA, 2001. IEEE Computer Society.
- [Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, October 2002.
- [Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
- [MASFPC11] Alberto Mendo Alonso, Javier Sobrino Fernández, Javier Pérez, and Yania Crespo. Evaluación de herramientas de detección y corrección de defectos de diseño. Technical Report 2011/02, GIRO research group, Departamento de Informática, Universidad de Valladolid, March 2011.
- [MBG06] Naouel Moha, Saliha Bouden, and Yann-Gaël Guéhéneuc. Correction of high-level design defects with refactorings. In Serge Demeyer, Stéphane Ducasse, Yann-Gaël Guéhéneuc, Kim Mens, and Roel Wuyts, editors, *Proceedings of the 7th ECOOP Workshop on Object-Oriented Reengineering*, July 2006.

- [MC03] R. Marticorena and Y. Crespo. Refactorizaciones de especialización sobre el lenguaje modelo MOON. Technical Report DI-2003-02, Departamento de Informática. Universidad de Valladolid, septiembre 2003. Available at <http://giro.infor.uva.es/docpub/marticorena-tr2003-02.pdf>.
- [MC08] Raul Marticorena and Yania Crespo. Dynamism in Refactoring Construction and Evolution – A Solution Based on XML and Reflection. In José Cordeiro, Boris Shishkov, AlpeshKumar Ranchordas, and Markus Helfert, editors, *ICSOFT 2008, Third International Conference on Software and Data Technologies, Porto, Portugal*, pages 214–219. INSTICC - Institute for Systems and Technologies of Information, Control and Communication, july 2008.
- [MCL05] Raúl Marticorena, Yania Crespo, and Carlos López. Soporte de métricas con independencia del lenguaje para la inferencia de refactorizaciones. In *X Jornadas Ingenieria del Software y Bases de Datos (JISBD 2005), Granada, Spain*, pages 59–66, September 2005.
- [MDBJ08] Olaf Muliawan, Bart Du Bois, and Dirk Janssens. Refactoring using JDT2MDR, an industrial based solution. In *4th International Workshop on Graph-Based Tools: The Contest (GraBaTs 2008)*, September 2008. <http://www.fots.ua.ac.be/events/grabats2008>.
- [Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, May 2007.
- [MGDLM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, January/February 2010.
- [MGLM⁺09] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, May 2009.
- [MGMD08] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien. A domain analysis to specify design defects and generate detection algorithms. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2008.
- [MHB07] Emerson R. Murphy-Hill and Andrew P. Black. High velocity refactorings in Eclipse. In Li-Te Cheng, Alessandro Orso, and Martin P. Robillard, editors, *ETX*, pages 1–5. ACM, 2007.
- [MHB08] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 421–430, New York, NY, USA, 2008. ACM.

- [MHVG08a] Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. Refactorings of design defects using relational concept analysis. In Raoul Medina and Sergei A. Obiedkov, editors, *ICFCA*, volume 4933 of *Lecture Notes in Computer Science*, pages 289–304. Springer, 2008.
- [MHVG08b] Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. Refactorings of design defects using relational concept analysis. In Raoul Medina and Sergei Obiedkov, editors, *4th International Conference on Formal Concept Analysis*, pages 289–304. Springer-Verlag, February 2008.
- [MJ10] Olaf Muliawan and Dirk Janssens. Model refactoring using MoTMoT. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:201–209, 2010. 10.1007/s10009-010-0147-9.
- [MKR06] Tom Mens, Günter Kniesel, and Olga Runge. Transformation dependency analysis – a comparison of two approaches. In Roger Rousseau, Christelle Urtado, and Sylvain Vauttier, editors, *Actes des journées Langages et Modèles à Objets, LMO’06. Nîmes, 22-24 mars*, pages 167–184. Hermès Lavoisier, 2006.
- [MLC05] Raúl Marticorena, Carlos López, and Yania Crespo. Parallel inheritance hierarchy: Detection from a static view of the system. In *6th International Workshop on Object Oriented Reengineering (WOOR)*, Glasgow, UK., page 6, July 2005. <http://smallwiki.unibe.ch/woor/workshopparticipants/>.
- [MLC06] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *7th ECCOP International Workshop on Object-Oriented Reengineering (WOOR)*, page 6, Nantes, France, July 2006.
- [MLCP07] Raul Marticorena, Carlos López, Yania Crespo, and Javier Pérez. Reuse based refactoring tools. In Michael Cebulla Danny Dig, editor, *Proceedings 1st Workshop on Refactoring Tools (WRT 07)*., pages 21–23. TU Berlin, Germany, July 2007. Bericht-Nr. 2007–8.
- [MMM⁺05] Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma: An integrated platform for quality assessment of Object-Oriented design. In *ICSM (Industrial and Tool Volume)*, pages 77–80, 2005.
- [Moh08] Naouel Moha. *DECOR : Détection et correction des défauts dans les systèmes orientés objet*. PhD thesis, Université des Sciences et Technologies de Lille; Université de Montréal, August 2008.
- [MPS01] A.M. Memon, M.E. Pollack, and M.L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, february 2001.
- [MSF⁺10] Naouel Moha, Sagar Sen, Cyril Faucher, Olivier Barais, and Jean-Marc Jézéquel. Evaluation of Kermeta for solving graph-based problems. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:273–285, 2010. 10.1007/s10009-010-0150-1.

- [MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
- [MT06] Hayden Melton and Ewan Tempero. Identifying refactoring opportunities by identifying dependency cycles. In *ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, pages 35–41, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
- [MTR07] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, September 2007.
- [Mun05] Matthew James Munro. Product metrics for automatic identification of “bad smell” design problems in java source-code. *Software Metrics, 2005. 11th IEEE International Symposium*, pages 9 pp.–, Sept. 2005.
- [MVDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July/August 2005.
- [MyT] MyTelly. Tom Talbott, <http://mytelly.sourceforge.net>.
- [NAI⁺03] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.
- [NB07] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007)*, volume 4581/2007 of *Lecture Notes in Computer Science*, pages 228–243. Springer, Heidelberg, June 2007.
- [NL06] Colin J. Neill and Phillip A. Laplante. Paying down design debt with strategic refactoring. *IEEE Computer*, 39(12):131–134, 2006.
- [OCBZ09] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *3rd International Symposium on Empirical Software Engineering and Measurement, ESEM'09*, pages 390–400, Washington, DC, USA, 2009. IEEE Computer Society.
- [OKAG10] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Numerical signatures of antipatterns: An approach based on b-splines. In *14th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, March 2010.
- [Opd92] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992. also Technical Report UIUCDCS-R-92-1759.

- [PC07a] Javier Pérez and Yania Crespo. Exploring a method to detect behaviour-preserving evolution using graph transformation. In Tom Mens, Kim Mens, Ellen Van Paesschen, and Maja D'Hondt, editors, *Proceedings of the Third International ERCIM Workshop on Software Evolution*, pages 114–122, Paris, France, October 2007. ERCIM.
- [PC07b] Javier Pérez and Yania Crespo. A refactoring discovering tool based on graph transformation. In Michael Cebulla Danny Dig, editor, *Proceedings 1st Workshop on Refactoring Tools (WRT 07)*, pages 19–20. TU Berlin, Germany, July 2007. Bericht-Nr. 2007–8.
- [PC09] Javier Pérez and Yania Crespo. Perspectives on automated correction of bad smells. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 99–108, New York, NY, USA, 2009. ACM.
- [PCHM10] Javier Pérez, Yania Crespo, Berthold Hoffman, and Tom Mens. A case study to evaluate the suitability of graph transformation tools for program refactoring. *International Journal on Software Tools for Technology Transfer*, pages 183–199, 2010.
- [PCML03] Félix Prieto, Yania Crespo, José Manuel Marqués, and Miguel A. Laguna. Applying formal concepts analysis to the construction and evolution of domain frameworks. In *6th International Workshop on Principles of Software Evolution (IWPSE 2003). Helsinki, Finland. ISBN 0-7695-1903-2.*, pages 139–148. IEEE Computer Society, sep 2003.
- [Pér05] Javier Pérez. Automated elaboration of refactoring plans. In *Preproceedings of the 1st Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE*, Braga, Portugal, July 2005.
- [Pér06] Javier Pérez. Overview of the refactoring discovering problem. In *Doctoral Symposium, 20th edition of the European Conference on Object-Oriented Programming (ECOOP 2006)*, July 2006.
- [Pér08] Javier Pérez. Enabling refactoring with HTN planning to improve the design smells correction activity. In *BENEVOL 2008 : The 7th BELgian-NEtherlands software eVOLution workshop*, December 2008.
- [PLMM11] Javier Pérez, Carlos López, Naouel Moha, and Tom Mens. A classification framework and survey for design smell management. Technical Report 2011/01, GIRO research group, Departamento de Informática, Universidad de Valladolid, March 2011.
- [PMD02] PMD. <http://pmd.sourceforge.net>, June 2002.
- [Pou] Pounder. Matthew Pekar, <http://pounder.sourceforge.net>.
- [PP07] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

- [PRT08] Javier Pérez, Olga Runge, and Gabriele Taentzer. Specifying and analyzing program refactorings with AGG. In *4th International Workshop on Graph-Based Tools: The Contest (GraBaTs 2008)*, September 2008. <http://www.fots.ua.ac.be/events/grabats2008>.
- [Pti] The Ptidej tool suite. Includes the DECOR method. The Ptidej team; École Polytechnique Montréal, Université de Montréal. <http://ptidej.dyndns.org>.
- [PVDSM10] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Automated planning for resolving model inconsistencies: A scalability study. In *International Workshop on Models Evolution ME 2010*, 2010.
- [Raț04] Daniel Rațiu. Memoria: A Unified Meta-Model for Java and C++. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timișoara, 2004.
- [Ree92] Jack Reeves. What is software design? *C++ Journal*, 2(2), 1992.
- [ReJ] ReJ. RevJava. <http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm>.
- [RFG05] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving evolvability through refactoring. *SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.
- [RH07] Joachim Hertzberg Ronny Hartanto. Augmenting jshop2 planning with owl. In *Proceedings of the 21th Workshop Planen, Scheduling und Konfigurieren, Entwerfen (PuK)*, Osnabrück, Germany, September 2007.
- [Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, April 1996.
- [Rob99] Donald Bradley Roberts. *Practical Analysis For Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- [Roo] Roodi. <http://roodi.rubyforge.org>.
- [Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume I: Foundations*. World Scientific, 1997.
- [RR08] A. Ananda Rao and K. Narendar Reddy. Detecting bad smells in object oriented design using design change propagation probability matrix. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, March 2008.
- [RS97] J. Rekers and Andy Schurr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.
- [RSG08] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*, pages 35–38, New York, NY, USA, 2008. ACM.

- [RSS⁺04] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: Smart analysis based error reduction. *ACM SIGSOFT Software Engineering Notes*, 29(4):243–251, 2004.
- [Rut] Kevin Rutherford. Reek. <http://wiki.github.com/kevinrutherford/reek>.
- [SCJ88] Sidney Siegel and N. John Castellan Jr. *Nonparametric statistics for the behavioral sciences*. McGraw-Hill Book Company, 2nd edition edition, 1988.
- [Sem07] SemmleCode. <http://semmle.com/semmlecode>, October 2007.
- [SGM00] Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of OO design quality and its automation? In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 154–162, Washington, DC, USA, 2000. IEEE Computer Society.
- [SGSM10] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, 99(27):52–57, Jul-Aug 2010.
- [She00] Thomas B. Sheridan. Function allocation: algorithm, alchemy or apostasy? *Int. J. Hum.-Comput. Stud.*, 52(2):203–216, 2000.
- [Sli05] Stefan Slinger. Code smell detection in Eclipse. Master’s thesis, Faculty of Electrical Engineering, Mathematics and Computer Science - Delft University of Technology, 2005.
- [SLMM99] Houari A. Sahraoui, Hakim Lounis, Walcélio L. Melo, and Hafedh Mili. A concept formation based approach to object identification in procedural code. *Automated Software Engineering*, 6(4):387–410, 1999.
- [SLT06] Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. A metric-based heuristic framework to detect object-oriented design flaws. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)IEEE*, 2006.
- [SPW⁺04] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, October 2004.
- [SR06] Martin Lippert Stefan Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, 2006.
- [SRK07] Daniel Speicher, Tobias Rho, and Günter Kniesel. Code analyses for refactoring by source code patterns and logical queries. In Michael Cebulla Danny Dig, editor, *Proceedings 1st Workshop on Refactoring Tools (WRT'07)*. TU Berlin, Germany, 2007. Bericht-Nr. 2007–8.
- [SS07a] G. Snelting and M. Streckenbach. Kaba: Automated refactoring for improved cohesion. In *1st Workshop on Refactoring Tools (WRT'07)* [DC07], pages 1–2.

- [SS07b] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring—does it improve software quality? *5th International Workshop on Software Quality*, 0:10, 2007.
- [SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38, Washington, DC, USA, 2001. IEEE Computer Society.
- [ST00] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions Programming Languages Systems*, 22(3):540–582, 2000.
- [STA] STAN. Available at <http://stan4j.com>.
- [Sty] StyleCop. <http://code.msdn.microsoft.com/sourceanalysis>.
- [Tae03] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.
- [TCC08] Nikolaos Tsantalis, Tsantalis Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. *CSMR 2008. 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, April 2008.
- [TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of the International Symposium Principles of on Software Evolution*, pages 157–167. IEEE Computer Society Press, 2000.
- [Tea09] Ptidej Team. DECOR, 2009. Available at <http://www.ptidej.net/research/decor>.
- [Tes08] Jean Tessier. Dependency Finder, 2008. Available at <http://depfind.sourceforge.net>.
- [Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.
- [TK04] Ladan Tahvildari and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 16(4-5):331–361, 2004.
- [TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 91–100, Washington, DC, USA, 2003. IEEE Computer Society.
- [TMMM07] Alban Tiberghien, Naouel Moha, Tom Mens, and Kim Mens. Répertoire des défauts de conception. Technical Report 1303, University of Montreal, 2007.

- [TR07] Adrian Trifu and Urs Reupke. Towards automated restructuring of object oriented systems. *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, pages 39–48, March 2007.
- [Tri08] Adrian Trifu. *Towards Automated Restructuring of Object Oriented Systems*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.
- [TSFB99] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in Object-Oriented designs: using reading techniques to increase software quality. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and applications*, pages 47–56, New York, NY, USA, 1999. ACM.
- [TSG04] Adrian Trifu, Olaf Seng, and Thomas Genssler. Automated design flaw correction in Object-Oriented systems. In Claudio Riva and Gerardo Canfora, editors, *proceedings of the 8th Conference on Software Maintenance and Reengineering*, pages 174–183. IEEE Computer Society Press, March 2004.
- [vDMvdBK01] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi and G. Succi, editors, *Proceedings of the Second International Conference on Extreme Programming*, 2001.
- [VKMG09] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*, pages 145–154, Los Alamitos, CA, USA, October 2009. IEEE Computer Society.
- [VRDBDR07] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.
- [Wak03] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [Web95] Bruce F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, 1st edition, February 1995.
- [Wei08] Erhard Weinell. Using PROGRES for graph-based program refactoring. In *4th International Workshop on Graph-Based Tools: The Contest (GraBaTs 2008)*, September 2008. <http://www.fots.ua.ac.be/events/grabats2008>.
- [WP05] Bartosz Walter and Blazej Pietrzak. Multi-criteria detection of bad smells in code with UTA method. In Hubert Baumeister, Michele Marchesi, and Mike Holcombe, editors, *XP*, volume 3556 of *Lecture Notes in Computer Science*, pages 154–161. Springer, June 2005.
- [WRH⁺00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöörn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

- [XS04] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In Frank Maurer and Günther Ruhe, editors, *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004)*, pages 123–128, June 2004.
- [ZHH⁺09] Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Hector Munoz-Avila. Learning htn method preconditions and action models from partial observations. In *Proceedings of the 21st international joint conference on Artificial intelligence*, pages 1804–1809, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

Appendix A

JSHOP2 Planning Language Specification

We compile here, for reference purposes, the description of the subset of the JSHOP2 language we have actually employed. This has been extracted from its original source [Ilg06].

A.1 Symbols

A symbol s can be composed of letters and digits, question marks, exclamations, hyphens and underscores. Function symbols are a special type of symbols. These are valid Java identifiers which are used to invoke external procedure calls. JSHOP2 defines the following types of symbols:

- Constant symbols, compound task symbols, predicate symbols: s
i.e.: some-constant; some-task; some-predicate
- Variable symbols: $?s$
i.e.: ?some-variable
- Primitive task symbols: $!s$
i.e.: !a-primitive-task
- Auxiliar tasks symbols: $!!s$
i.e.: !!an-auxiliary-task

A.2 Terms

Terms are the most basic entities in the language. Simple terms are numerical constants, and variable and constant symbols. Composed terms are list terms and call terms.

- List terms: $(T_1 \ T_2 \ \dots \ T_n)$
Where each T_i is a term. An special list term `-nil-` represents the empty list.
i.e. (package 10079 java_dot_lang); (arg1 arg2 arg3); nil

- Call terms: `(call F T1 T2 ... Tn)`

Where F is a function symbol and each T_i are terms. When a call term is evaluated, the external JAVA procedure F is invoked and each T_i term is passed as an argument. After being evaluated, the call term is substituted by the result term produced by the external procedure. External procedures can be built-in procedures includes in JSHOP2 or user-defined pocedures. The available built-in procedures are: `<`, `<=`, `=`, `>`, `>=`, `!=`, `+`, `-`, `*`, `/`, `^`. User-defined procedures have to be implemented as JAVA classes.

i.e. `(call GetRandom nil); (call UserQuery please-suggest-a-target-class-for ?source-class)`

A.3 Logical atoms and logical expressions

These constructions are the afirmations over the state of the world that can be evaluated. The simpler logical expressions are logical atoms and call expressions.

- Logical atom: `(P T1 T2 ... Tn)`

Where P is a predicate symbol and T_i are terms.

i.e. `(package 10079 java_dot_lang)`

A logical expression is any expression that can be evaluated to a boolean value. Logical atoms is a logical expression that will be evaluated to `true` if it exists in the current state of the world or can be derived from it and to `false` otherwise. Call expressions are logical expressions that are evaluated to `false` if the external procedure returns `nil` or the empty list and to `true` otherwise.

- And: `([and] [L1 L2 ... Ln])`

It evaluates to `true` if all the logic expressions L_i evaluate to `true` and `false` otherwise. The `and` symbol is optional. An empty logical expression `()`, `(and)` or `nil` evaluates to `true`.

- Or: `(or L1 L2 ... Ln)`

It evaluates to `true` if at least one of the logic expressions L_i evaluate to `true` and `false` otherwise.

- Not: `(not L)`

It evaluates to `true` if there is not a valid substitution that makes L evaluate to `true` and `false` otherwise.

- Imply: `(imply Y Z)`

It evaluates to `true` if there is not a valid substitution that makes Y evaluate to `true`, if there is a valid substitution that makes Y and Z simultaneously `true`, and `false` otherwise.

- For all: `(forall V Y Z)`

It evaluates to `true` if for all the possible substitutions for the variables in V which makes Y evaluate to `true`, Z also evaluates to `true`. It evaluates to `false` otherwise.

- Assign: `(assign V T)`

The value bounded to Z is bounded to V . It evaluates to `true`.

A.4 Logical preconditions

Logical preconditions are logical expressions that control whether a particular method or operators should be included in a plan or not.

- Regular precondition: L

Where L is a logical precondition.

eg.: (superclass a-class a-superclass)

- First satisfier precondition: (:first L)

Where L is a logical precondition.

eg.: (:first (superclass ?class a-superclass))

will be evaluated until a valid substitution for `?class` is found or until none can be found. No other valid substitution will be tried even if the first one does not lead to a valid plan.

A.5 Axioms

Axioms are Horn-Clause-like expressions that can be used to derive additional knowledge from the system's state.

- Axiom: (:– H [L_1] T_1 [L_2] T_2 ... [L_n] T_n)

where H is the axiom's head, each L_i is an optional labels and each T_i is an axiom branch. The axiom's head H is a logical atom. It will be evaluated to true if T_j is evaluated to true and every T_i , with $i < j$ is evaluated to false. An axiom branch T_i will be evaluated and an unification attempt will be carried on only if all the precedent branches have failed.

eg.: (:– (contains-member ?class ?member)
 member-is-field
 (contains-field ?class ?member)
 member-is-method
 (contains-method ?class ?member)
)

A.6 Task atoms and task lists

Tasks atoms are used to “declare” and identify tasks. They can appear in the definition of a task method as their “header” or be “invoked” in the method's body as the method's task decomposition.

- Task atom: ([[!]] S T_1 T_2 ... T_n)

where S is a task symbol that can be interpreted as the task's name and T_i are terms, that are mostly used as the task's arguments. A task is identified by its name and the number of terms in its definition. A primitive task name is written with a primitive task symbol: `!S`. An auxiliary task name is written with an auxiliary task symbol: `!!S`. A compound task name is written with a compound task symbol: `S`. A task atom is also a task list.

- Task list: (`[[:unordered]` $T_1 T_2 \dots T_n$);

where each T_i is a task list. The optional `unordered` keyword specifies that any T_i ordering should be valid. This leaves the planner the freedom to interleave the tasks in those task lists during the planning process.

A.7 Operators

Operators are the atomic tasks that can be applied to a system in order to change its state.

- Operator: (`:operator` $H P D A [C]$)

where H is a primitive task term, P is a precondition, D is a list of negative effects, A is a list of positive effects and C is the optional cost of applying the operator. Effects in D and A are expressed as a list of logical atoms. The operator is identified by its head H . The precondition P specifies the condition under which the operator can be applied. The lists of effects define how the operator changes the system's state by adding or removing the logical terms included in A or D respectively.

A.8 Methods

Methods are specifications of how to achieve a compound tasks by decomposing it into simpler tasks. If the method's precondition is hold in the current state of the system, the compound task can be achieved by performing the list of tasks defined in the method.

- Method: (`:method` $H [L_1] P_1 TL_1 [L_2] P_2 TL_2 \dots [L_n] P_n TL_n$)

where H is a compound task term, and each set $[L_i] P_i TL_i$ is a different method branch, where L_i is the optional label, P_i is the precondition and TL_i is task list of that branch. Alternative decompositions within a method definition are visited in sequential order. The preconditions are considered in the given order. The branch i is only attempted if the precondition of the branch $i-1$ is not met. Even if the branch $i-1$ does not lead to a valid plan, the branch i is not attempted unless precondition $i-1$ fails.

A.9 Planning domains

A planning domain is used to compile and specify all the domain knowledge available for a given problem. A planning domain is comprised of a task network, specified with operators and methods, and axioms, which allow to derive additional knowledge from the system's state.

- Planning domain: (`defdomain` $N (D_1 D_2 \dots D_n)$),

where the symbol N is the name of the planning domain and D_i are operators, methods or axioms.

A.10 Planning problems

A planning problem is defined by a domain specification, the initial state of the system and the goal to achieve. In HTN planning the goal is given as a task list. In JSHOP2 it is possible to formulate and run several planning problems over a shared domain definition with a single construct and file.

- Planning problem: `(defproblem PN DN L1 T1 L2 T2 .. Ln Tn)`

where the symbol `PN` is the name of the planning problem, `DN` is the name of the domain, and each `Li Ti` is a separate planning problem where `Li` is a list of ground logical atoms representing the system's initial state and `Ti` the goal task list for the problem `i`.

Appendix B

Translation of the refactoring strategy language

Refactoring strategies can be written with the small language we have provided in Chapter 4. We have not developed a compiler or translator for the refactoring strategies specification language. Nevertheless, this Appendix demonstrates how each refactoring strategy construct can be translated into the JSHOP2 language so they can be added to a domain specification for the planner. The translations are displayed with template samples of refactoring strategies, shown on the left-hand side, and their matching JSHOP2 code on the right-hand side.

B.1 Strategy Definitions

```
Strategy goal name (arg1, ..., argn)
  precondition
    PREC
  body
    STEPS
end
```

```
(:method (goal name ?arg1 ... ?argn)
  (PREC)
  (STEPS)
)
(method (try goal name ?arg1 ... ?argn)
  ()
  (
    (goal name ?arg1 ... ?argn)
  )
)
(method (try goal name ?arg1 ... ?argn)
  ()
  ()
)
```

A Refactoring Strategy is straightforward translated as a JSHOP2 method. The strategy's precondition and body are translated into a method's single branch as the precondition and task decomposition task list of this branch. Along with each strategy two additional methods have to be created in order to enable the `try` invocation construct. The first `try` method attempts to apply the involved strategy as it is. In the case this method fails, the second `try` method will be carried out by the planner. The second `try` method has an empty precondition and task decomposition list, therefore it succeeds trivially. Note that all variables are preceded with a question mark (?) in the JSHOP2 language.

B.2 Sequences of steps

Blocks, or sequences of steps, have to be processed and translated in a recursive way. In order to be translated into a JSHOP2 domain, a block or a sequence of steps has to be split in different fragments and then translated into several methods. This description will be first focused on how the block fragments are translated. The explanation on how the blocks of sentences are split into fragments is performed afterwards.

<pre> Strategy goal name (arg1, ..., argn) precondition PREC body S1 S2 S3 S4 end </pre>	<pre> (:method (goal name ?arg1 ... ?argn) (PREC) ((M1 ARGS1) (M2 ARGS2) (M3 ARGS3) (M4 ARGS4))) (:method (M1 ARGS1) (P1) (S1)) (:method (M2 ARGS2) (P2) (S2)) (:method (M3 ARGS3) (P3) (S3)) (:method (M4 ARGS4) (P4) (S4)) </pre>
--	---

Two different procedures can be used to split and translate these fragments of step sequences: a hierarchical, or a chained way. The hierarchical procedure is shown above. Each block fragment is transformed into a separated JSHOP2 method M_i , whose task list contains the translation of each S_i . The main strategy is translated as a “top” method from which these methods are invoked. The “top” method’s task decomposition list contains the invocations of each M_i .

The translation of each method M_i depends on the recursive translation of each block fragment S_i . The list of arguments of each method $ARGS_i$ will declare the variables that may have been bound in the top method and are needed in each block fragment method M_i . The variables to pass are all the variables in the top method’s argument list or precondition that appear in the respective S_i . The preconditions P_i of each M_i will be obtained from the recursive translation of each block fragment S_i .

The chained procedure for translating a block of steps is displayed below. The translation procedure is similar to the previous one, although it differs on how the block fragment’s methods M_i are connected and invoked. With this procedure, the “top” method’s task decomposition list does only include the invocation of the initial method M_1 . Then, each M_i method includes the invocation of the next M_i in its task decomposition list. With this procedure, the variables needed for each M_i are passed through the invocation chain from a method to another. Each argument list $ARGS_i$ will contain the variables needed by M_i and also the variables needed by the next methods in the chain (M_{i+1}), that may have been bounded in the previous method (M_{i-1}) or in the “top” method.


```

Strategy goal name (arg1, ..., argn)
  precondition
    PREC
  body
    S1
    S2
    S3
    S4
end

```

```

(:method (goal name ?arg1 ... ?argn)
  (PREC)
  ( (M1 ARGS1) )
)
(:method (M1 ARGS1)
  (P1)
  (S1
    (M2 ARGS2) )
)
(:method (M2 ARGS2)
  (P2)
  (S2
    (M3 ARGS3) )
)
(:method (M3 ARGS3)
  (P3)
  (S3
    (M4 ARGS4) )
)
(:method (M4 ARGS4)
  (P4)
  (S4)
)

```

As for splitting a block of steps into a set of fragments, the kind of steps involved has to be taken into account. A step can be either a query, an invocation or a compound step. In order to translate it, a sequence of steps should be split into the two following types of fragments: sequences of queries followed by invocations and compound steps. Each block fragment S_i will only contain query-invocation sequences or individual compound steps. An example of this is shown below:

```

query 1
invocation 1
invocation 2
query 2
invocation 3
compound step 1
invocation 4
compound step 2
compound step 3

```

```

S1:
  query 1
  invocation 1
  invocation 2
S2:
  query 2
  invocation 3
S3:
  compound step 1
S4:
  invocation 4
S5:
  compound step 2
S6:
  compound step 3

```

It should be noticed that, after splitting a block of steps and obtaining the basic structure by following the procedures described above, the translation of each S_i sequence of steps may have to be continued and performed recursively. This is needed when S_i represent a compound step and contains a nested block of steps. These compound steps can be either loop control structures, alt structures and unordered blocks. The translation of each particular fragment is described later on.

B.3 Managing variables between methods

Variables are bound to values exclusively in query invocations. Moreover JSHOP2 method's arguments are passed "by value", therefore a binding taking place within a method's precondition is not visible from outside that method, unless the variable is passed as an argument or its bound value is stored to the system's state. While translating a refactoring strategy, this has to be taken into account. We have to keep track of the variables that may have been bound in a JSHOP2 method and how they can be accessed from other methods where they are needed.

Variables are communicated through methods, either by passing them as arguments or by storing them in the system's state as persistent variables. The following rules have to be followed.

1. The initial variables' scope is a strategy.
2. Once a variable has been bound, its binding, or the variable attached value, can be directly accessed from within their scope or otherwise from the factbase (as persistent variables).
3. When generating/translating a fragment of a strategy into a JSHOP2 method, the accessibility of the needed variables should be checked. If the variable bindings are accessible from the outer scope, the variables should be passed as arguments.
4. If the variables are not accesible, they should be gathered with a persistent variable query. These queries should be included at the beginning of a precondition.
5. All variables used in a fragment that is going be translated, should be either passed as arguments or gathered with persistent variables queries.
6. All the variables that may have been bound within a certain scope, that is, all the variables within the precondition section of a JSHOP2 method, should be propagated to the outer scope.
7. If the strategy fragments are translated and chained in parallel, the variables used in the fragment should be recalled with persistent variables queries in the JSHOP2 method's precondition section.
8. If the strategy fragments are translated and chained in cascade, the variables used in the fragment should be passed as arguments.

Persistent variables are managed with the following refactoring planning domain elements:

- Queries:

- (rp-var ?varname ?var-value)

- Operators:

- (add-rpvar ?varname ?var-value)
 - (del-rpvar ?name ?value)
 - (rep-var ?name ?old-value ?new-value)
 - (del-all-rpvar ?name)

B.4 Managing lists

The auxiliary queries for managing lists that are used for translating refactoring strategy specifications into the JSHOP2 refactoring planning domain are listed below.

- (get-head ?item ?list)
- (get-rest- ?rest ?list)
- (member ?item ?list)
- (remove-member ?item ?list ?new-list)
- (add-member ?item ?list ?new-list)

B.5 Queries, invocations and boolean expressions

apply name (arg1, ..., argn)	(name ?arg1 ... ?argn)
apply goal name (arg1, ..., argn)	(goal name ?arg1 ... ?argn)
try name (arg1, ..., argn)	(try name ?arg1 ... ?argn)
try goal name (arg1, ..., argn)	(try goal name ?arg1 ... ?argn)
queryname (arg1, ..., argn)	(queryname ?arg1 ... ?argn)
(cond)	(cond)
not cond	(not cond)
cond1 or cond2	(or cond1 cond2)
cond1 or cond2 or ... or condN	(or cond1 cond2 ... condN)
cond1 and cond3	(and cond1 cond2)
	(cond1 cond2)
cond1 and cond2 and ... and condN	(and cond1 cond2 ... condN)
	(cond1 cond2 ... condN)

Invocations are translated straightforwardly. Apply invocations are translated as task atoms which match either a transformation, a refactoring or a strategy by referencing their names or goal-name set. In the case of the try invocations, the reference to the corresponding task is preceeded with the try term. Queries are translated as invocations of logic expressions. Their arguments are translated as the expression arguments and the query name is used as the first term of the predicate.

Regarding boolean expressions, they are translated by transforming them into LISP-like lists. A condition is written as a list enclosed in parenthesis, with the boolean operator as the first element of the list and the operands as the rest of the elements of the list. If the original condition present several consecutive boolean operations of the same kind, they can be translated as a single list, with a single operator and all the operands within the list. The and operator is optional in JSHOP2, so in the case of conjunctions, it can be omitted altogether.

As described in B.2, blocks or sequences of steps, have to be fragmented and transformed into several JSHOP2 methods. Fragments of queries and invocations are translated by transforming each fragment into a method whose precondition contains the fragment's queries and whose task decomposition list gathers the fragment's apply and try invocations. An example from B.2 is

used to show how sequences of queries and invocations are translated and the matching JSHOP2 methods are displayed below.

query 1	S1:	(:method (M1 ARGS1)
invocation 1	query 1	(query 1)
invocation 2	invocation 1	((invocation 1)
query 2	invocation 2	(invocation 2))
invocation 3	S2:)
compound step 1	query 2	(:method (M2 ARGS2)
invocation 4	invocation 3	(query 2)
compound step 2	S3:	((invocation 3))
compound step 3	compound step 1)
	S4:	; translation of M3
	invocation 4	
	S5:	(:method (M4 ARGS4)
	compound step 2	()
	S6:	((invocation 4))
	compound step 3)
		; translation of M5
		; translation of M6

B.6 Variables and literals

variable	?variable
"string literal"	string-literal
number	number

Variables and literals are easily translated into JSHOP2, albeit with some limitations regarding string literals. Refactoring strategies' variables are simply translated by preceeding their symbol with a question mark (?). The JSHOP2 language does not define any syntax structure to define or enclose string literals, therefore they have to be transformed into allowed JSHOP2 symbols. Non-permitted characters have to be either removed from the string or transformed. For example, we suggest to transform space characters into dash characters, as seen in the example above. A valid symbol has to begin wit a letter or an underline and can contain letters, digits, question marks, exclamation points, hyphens and underlines.

B.7 Alternatives

<pre> if COND1 then STEPS1 elseif COND2 then STEPS2 else STEPS3 end </pre>	<pre> ;; if invocation (if-xx ARGS) ;; if definition (:method (if-xx ARGS) (COND1) (STEPS1) (COND2) (STEPS2) () (STEPS3)) </pre>
--	--

There are two different types of alternative constructs: the regular one and the non-deterministic alternative. The regular alternative construct – *if-then-else* – is translated into a single JSHOP2 method with as many branches as the original alternative. The branches in the *if-then-else* alternative are matched by the method’s branches appearing in the same order. The condition of each branch in the *if-then-else* construct is translated as the precondition of the matching branch in the JSHOP2 method. The block of steps nested in the *if-then-else* construct is transformed into the task decomposition list of the method’s branch. These steps should be recursively translated as needed. The *else* part in the original structure is translated into the last method’s branch, whose precondition is empty and, therefore, trivially true.

The JSHOP2 method representing the *if-then-else* statement has to be named with a symbol with the prefix *if-* and a numeric suffix, so that the resulting symbol provides a unique name for the method in the refactoring planning domain. The *if-then-else* construct in the original sequence of steps should be then replaced by the invocation of the *if* method. The method’s arguments *ARGS* should contain all the variables appearing in the method that are accessible from the original sequence of steps’ scope.

All the variables used in each branch precondition of the method, may have been bound, therefore they have to be propagated. If the blocks of steps are being translated in a hierarchical way, the variables should be stored as persistent variables using the *store-rpvar* method. If the blocks of steps are being translated in a chained way, the next fragment’s method should be invoked with the last task in the task decomposition list of all the branches, and the variables should be passed as arguments.

The other kind of alternative construct is the non-deterministic alternative. it is translated in a similar way as the regular alternative, although each *alt* branch is translated into a separated JSHOP2 method. Each branch condition is translated into the precondition of each matching method. An *alt* branch with an empty condition will produce a method with an empty precondition. All methods share the same name and list of arguments and the original *alt* construct in the original sequence of steps should be replaced by an invocation of the task these methods implement. The name of this methods is a symbol with the prefix *alt-* and a numeric suffix, so that the resulting symbol provides a unique name for the method in the refactoring planning domain. The same procedures regarding how the bound variables have to be propagated into and from the methods that were described for the regular alternatives, apply to the *alt* non-deterministic construct. An example of *alt* translation is shown below.

<pre> alt branch COND1 body STEPS1 branch COND2 body STEPS2 branch body STEPS3 end </pre>	<pre> ;; alt invocation (alt-xx ARGS) ;; alt definition (:method (alt-xx ARGS) (COND1) (STEPS1)) (:method (alt-xx ARGS) (COND2) (STEPS2)) (:method (alt-xx ARGS) () (STEPS3)) </pre>
---	--

B.8 Loops: while

<pre> while COND loop VARS STEPS end </pre>	<pre> ;; while invocation (while-xx ARGS) ;; while definition (:method (while-xx ARGS) (:first (COND)) (STEPS (while-xx ARGS))) () ; exit loop ()) </pre>
---	--

A `while` loop is translated with a single method that contains two branches. The first branch of the method is used for translating the loop's elements, while the second branch has an empty precondition and an empty task decomposition list that works as the loop's exit branch. The loop's stay condition is translated as the first branch's precondition. The block of steps in the loop's body is translated into the first branch's task decomposition list. These steps should be recursively translated if needed. The last task in the first branch's decomposition list should be a recursive invocation of the loop's translated method.

Variable bindings taking place within the loop's method precondition are discarded between loop iterations by default. The variables in the `loop VARS` section should be kept between iterations. In order to translate it, the variables in `VARS` should be included in the `ARGS` argument list.

The method representing the `while` statement has to be named with a symbol with the prefix `while-` and a numeric suffix, so that the resulting symbol provides a unique name for the method in the refactoring planning domain. The `while` statement in the original sequence of steps should be then replaced by the invocation of the method. The method's arguments `ARGS` should contain all the variables appearing in the method's steps that are accessible from the

original sequence of steps' scope.

If the blocks of statements are being translated in a hierarchical way, queries for recalling the needed persistent variables that may have been previously stored, should be inserted as the first queries in the method's precondition. Also, if it is necessary to propagate the value of those variables that might have been bounded in the loop's body, the `store-rpvar` methods needed to save the bindings should be included in the task decomposition list of the exit branch.

If blocks of steps are being translated in a chained way, the invocation of the next block fragment should be included in the task decomposition list of the second branch. Also, the list of arguments `ARGS` should include all the variables that have to be passed to the following block fragments.

The translation of regular and non-deterministic loops is almost identic. Although different kind of JSHOP2 preconditions should be used. For a regular loop the first branch's precondition has to be a `first` precondition. For a non-deterministic loop, that allows the planner to check different iteration orderings, the first branch's precondition is a regular precondition.

```
while COND
  unordered loop VARS
  STEPS
end
```

```
;; while invocation
(while-xx ARGS)

;; while definition
(:method (while-xx ARGS)
  (
    (COND)
  )
  (
    STEPS
    (while-xx ARGS)
  )
  () ; exit loop
  ()
)
```

B.9 Loops: foreach

```
foreach V in VARS
  loop LVARs
  STEPS
end
```

```
;; foreach invocation
(foreach-xx ARGS VARS)

;; foreach definition
(:method (foreach-xx ARGS VARS)
  (:first
    (get-head V VARS)
    (get-rest REST VARS)
  )
  (
    STEPS
    (foreach-xx ARGS REST)
  )
  () ; exit loop
  ()
)
```

A `foreach` loop, with an explicit collection, is translated with a single `JSHOP2` method with two branches. The template above, describes a `foreach` loop in which a variable `v` is bound to each element of the list `VARS` in each iteration of the loop. The `VARS` list of elements to iterate over, should be passed as arguments of the `foreach` method. The first method branch is used to translate the loop's elements while the second branch is used to exit the loop. This second branch contains an empty precondition, that is trivially true, and an empty task decomposition list. The first branch's precondition should contain the necessary queries to control the iteration over the different elements of the `VARS` list. The first element in the list is bound to the variable `v`, the rest of the list is bound to the list `REST`. The `STEPS` in the loop's body are translated as tasks in the task decomposition list of the `foreach` method. This steps should be recursively translated if needed. In order to implement the loop, a recursive invocation of the `foreach` method is include as the last task in the first branch's task decomposition list. The `REST` list is passed to the invocation of the next loop's iteration as the collection to iterate over.

In the same way as `while` loops, variable bindings taking place within the loop's are discarded between loop iterations by default. The variables in the `loop VARS` section should be kept between iterations. In order to translate it, the variables in `LVARS` should be included in the `ARGS` argument list.

The method representing the `foreach` statement has to be named with a symbol with the prefix `foreach-` and a numeric suffix, so that the resulting symbol provides a unique name for the method in the refactoring planning domain. The `foreach` statement in the original sequence of steps should be then replaced by the invocation of the method. The method's arguments `ARGS` should contain all the variables appearing in the method's steps that are accessible from the original sequence of steps' scope.

If the blocks of statements are being translated in a hierarchical way, queries for recalling the needed persistent variables that may have been previously stored, should be inserted as the first queries in the first branch's precondition. Also, if it is necessary to propagate the value of those variables that might have been bounded in the loop's body, the `store-rpvar` invocations needed to save the bindings should be included in the task decomposition list of the first branch, between the translation of the loops' `STEPS` and the invocation of the next loop's iteration.

If blocks of steps are being translated in a chained way, the invocation of the next block fragment should be included in the task decomposition list of the second branch. Also, the list of arguments `ARGS` should include all the variables that have to be passed to the following block fragments.

The translation of regular and non-deterministic `foreach` loops is almost identic. Although they differ in how the `VARS` list to iterate its managed. For a regular loop the translation has already been described. For a non-deterministic loop, the queries in the first branch's precondition are used to randomly bound a member of `VARS` to a variable `v`. The `REST` list is obtained by removing this element `v` from the `VARS` list. The example templates are shown below.


```

foreach V in VARS
  unordered loop LVARs
    STEPS
  end

```

```

;; foreach invocation
(foreach-xx ARGS VARS)

;; foreach definition
(:method (foreach-xx ARGS VARS)
  (
    (member V VARS)
    (remove-member V VARS REST)
  )
  (
    STEPS
    (foreach-xx ARGS REST)
  )
  () ; exit loop
  ()
)

```

B.10 Loops: foreach

```

foreach VARS satisfying COND
  loop LVARs
    STEPS
  end

```

```

;; foreach invocation
(foreach-xx ARGS ())

;; foreach definition
(:method (foreach-xx ARGS C_VARS)
  (:first
    (
      COND
      (not (member VARS C_VARS))
      (add-member VARS C_VARS C_VARS2)
    )
  )
  (
    STEPS
    (foreach-xx ARGS C_VARS2)
  )
  () ; exit loop
  ()
)

```

The translation of a `foreach` loop that iterates over an implicitly defined collection is quite similar to the translation of the `foreach` loop that iterates over an explicit collection.

The template above, describes a `foreach` loop in which each distinct binding for the variable tuple `VARS` that satisfies the condition `COND`, is used at each loop's iteration. As in the other types of loops, the list of variables `LVARs` specify those variables whose bindings have to be kept between loop iterations.

The condition `COND` that has to be evaluated at each loop's iteration is translated into the first branch's precondition. A list of variable tuples `C_VARS` is used to keep track of the bindings of the variables in `VARS` that have already been checked. The queries that manage these list should be added at the end of the first branch's precondition. Once the condition `COND` has been evaluated within the `JSHOP2` method, the control queries check whether this binding has already been checked, and if it is not, the variable tuple defined by `VARS` is added to the control list `C_VARS`.

The new control list CVARS2 is passed to the next loop iteration. The initial invocation of the foreach method should include an empty list, that serves as the initial CVARS control list.

The translation of regular and non-deterministic foreach loops that iterate over implicit conditions are almost identical. Although different kind of JSHOP2 preconditions should be used. For a regular loop the first branch's precondition has to be a `first` precondition. For a non-deterministic loop, that allows the planner to check different iteration orderings, the first branch's precondition is a regular precondition. The example template for non-deterministic foreach loops is displayed below.

<pre> foreach VARS satisfying COND unordered loop LVARs STEPS end </pre>	<pre> ;; foreach invocation (foreach-xx ARGS ()) ;; foreach definition (:method (foreach-xx ARGS CVARS) (COND (not (member VARS CVARS)) (add VARS CVARS CVARS2)) (STEPS (foreach-xx ARGS CVARS2)) () ; exit loop ()) </pre>
--	--

B.11 Unordered steps

```

unordered
  Step1
  Step2
  Step3
end

```

```

;; unordered block invocation
(unordered-xx ARGS)
(unordered-xx ARGS)
(unordered-xx ARGS)

;; unordered block definition
(:method (unordered-xx ARGS)
  (;step1
    (not (rp-var unordered-xx-control step1))
  )
  (
    Step1
    (add-rpvar unordered-xx-control step1)
  )
)
(:method (unordered-xx ARGS)
  (;step2
    (not (rp-var unordered-xx-control step2))
  )
  (
    Step2
    (add-rpvar unordered-xx-control step2)
  )
)
(:method (unordered-xx ARGS)
  (;step3
    (not (rp-var unordered-xx-control step3))
  )
  (
    Step3
    (add-rpvar unordered-xx-control step3)
  )
)

```

A block of unordered steps is translated into the JSHOP2 refactoring planning domain by transforming the steps into a set of different methods sharing the same task symbol. The methods representing each step have to be named with a symbol that includes the prefix `unordered-` and a numeric suffix, so that the resulting symbol provides a unique name for this unordered block in the refactoring planning domain. The `unordered` statement in the original sequence of steps should be then replaced by a sequence of identical invocations of these methods, with as many identical invocations as steps in the unordered block or methods in the translation. The methods' arguments `ARGS` should contain all the variables appearing in all the steps in the unordered block. Independently of a certain variable being used or not for a particular step, all the `ARGS` lists of arguments should be the same.

Each unordered method, is used to translate each step in the original unordered block that have to be planed once and only once. In order to keep track of this, a persistent variable named after the methods name is used to control whether each step has been already planned or not. The persistent variable is stored with multiple values: `-step1, step2, ..., stepn-`, to specify that the corresponding step has already been planned. Within the precondition of each method, a persistent variable query is used to verify that the method has not been planned yet. The original steps are included in the task decomposition list of the methods and should be recursively translated if needed.

If the blocks of statements are being translated in a hierarchical way, queries for recalling the needed persistent variables that may have been previously stored, should be inserted as the last queries in the methods' precondition, after the query that verifies the control persistent variable of the unordered block. Also, if it is necessary to propagate the value of those variables that might have been bounded within the method, the `store-rpvar` methods needed to save the bindings should be included in the task decomposition list of each method, after the task that stores the persistent variable that controls the unordered block.

If blocks of steps are being translated in a chained way, the invocation of the next block fragment should be included as the last task in each method's task decomposition list. As in the previous types of control structures, the list of arguments `ARGS` should include all the variables that have to be passed to the following block fragments.

Appendix C

The Refactoring Planning Domain

This appendix lists the main elements defined in the refactoring planning domain.

C.1 Notation

The refactoring domain elements in this appendix are listed using PROLOG-like notation. This notation is used refactoring strategy specification language. Some parts of the domain, like system queries, have been written with this notation and are translated automatically in order to be fed to the planner. External procedures have been written in JAVA. The rest of the refactoring planning domain –operators and methods representing strategies, refactorings and the rest of transformations– have been written in the JSHOP2 language. Nevertheless, they are listed here using the PROLOG notation, since they would be mostly used with the refactoring strategy specification language.

The code fragments below illustrate how the PROLOG-like notation is translated to the JSHOP2 LISP-like notation.

<code>[]</code>	<code>() NIL</code>
<code>Variable</code>	<code>?Variable</code>
<code>symbol(Variable, literal, ``string literal``)</code>	<code>(symbol ?Variable literal string- literal)</code>
<code>goal strategy (argument)</code>	<code>(goal strategy ?argument)</code>

C.2 Refactoring Strategies

- `move-method all-sts (method, tgt-class, reference)`
- `remove-feature-envy all-sts (method)`
- `remove-data-class all-sts (class)`

C.3 Refactorings

- `encapsulate-field (field, getter-name, setter-name)`
- `move-method (method, tgt-class, reference, delegate)`
- `remove-class (class)`
- `remove-field (field)`
- `remove-method (method)`
- `rename-field(field, new-name)`
- `rename-local-variable (variable, new-localvar-name)`
- `rename-method (method, new-method-name)`
- `rename-parameter(parameter, new-parameter-name)`

C.4 Operators

- Add a system element:

```
(:operator (!add ?term)
  ()
  ()
  (?term)
  0      ; cost 0
)
```

- Deletes a system element:

```
(:operator (!del ?term)
  ()
  (?term)
  ()
  0      ; cost 0
)
```

- Replaces a system element:

```
(:operator (!rep ?old-term ?new-term)
  ()
  (?old-term)
  (?new-term)
  0      ; cost 0
)
```

C.5 External Java Procedures

All external procedures takes list of terms as inputs.

- `Concatenate(?list)`
Takes a list of symbols or ground variables and returns a symbol that is the concatenation of all them.
- `Debugger(?list)`
Takes a list of symbols and prints them to standard output.
- `GetRandom(?list)`
Returns a random integer between 1 and 2^{31} .
- `LesserVisibility(?list)`
Takes two terms and returns `true` if the first has a lesser visibility than the second one.
- `GreaterVisibility(?list)`
Takes two terms and returns `true` if the first has a greater visibility than the second one.
- `IsEqual(?list)`
Takes a lists of terms and returns *true* if all the terms are equal.
- `IsGround(?list)`
Takes a list of terms and returns *true* if all the terms are ground.
- `UserQuery(?list)`
Takes a list of terms as input. The input is interpreted as a message. The message is shown to the user and a user input is requested. After the user introduces a response, the procedure returns the information entered by the user as a constant term.

C.6 System Queries

- `fqn(Entity, (Package, Class, Field))`
- `fqn(Entity, (Package, Class, Method))`
- `fqn(Entity, (Package, Class))`
- `field-fqn(Field, Pname, Cname, Fname)`
- `method-fqn(Method, Pname, Cname, Mname)`
- `class-fqn(Class, Pname, Cname)`
- `is-accessor(Method, Field)`
- `is-accessor(Package, Class, Method, AccessedField)`

- `is-setter(Package, Class, Method, Field)`
- `is-setter(Method, Field)`
- `is-getter(Package, Class, Method, Field)`
- `is-getter(Method, Field)`
- `call-to-class-from(CallerMethod, CalledMethod, Class, Call)`
- `access-to-class-from(AccessMethod, Accessedfield, Class, Access)`
- `client-method-of-class(Method, Class)`
- `declares-member(Package, Class, Method)`
- `declares-field(Package, Class, Field)`
- `declares-method(Package, Class, Method)`
- `declares-member(Class, Member)`
- `declares-field(Class, Field)`
- `declares-method(Class, Method)`
- `extends(Class, SuperClass)`
- `inherits(Class, SuperClass)`
- `contains-member(Package, Class, Method)`
- `contains-member(Class, Member)`
- `contains-field(Class, Field)`
- `contains-method(Class, Method)`
- `inherits-member(Class, Method)`
- `inherits-field(Class, Field)`
- `inherits-method(Class, Method)`
- `inherits-member-from(Class, Method, SuperClass)`
- `inherits-field-from(Class, Field, SuperClass)`
- `inherits-method-from(Class, Method, SuperClass)`
- `implements-interface(Class, Interface)`
- `explicit-visibility(Entity, Vis)`
- `implicit-visibility(Entity, package)`

- `visibility(Entity, Vis)`
- `field-visibility(Package, Class, Field, Vis)`
- `method-visibility(Package, Class, Method, Vis)`
- `is-static(ContainerPKG, ContainerClass, MethodName)`
- `is-static(Entity)`
- `is-method-void(Package, Class, Method)`
- `is-constructor(Method)`
- `get-method-parameters(Package, Class, Method, ParamList)`
- `get-method-single-parameter(Package, Class, Method, Param)`
- `get-feature-type(Package, Class, Field, Type)`
- `get-feature-type(Package, Class, Method, Type)`
- `get-field-type(Package, Class, Field, Type)`
- `get-method-type(Package, Class, Method, Type)`
- `get-parameter-type(Package, Class, Method, Param, Type)`
- `first-parameter(Method, Param)`
- `type-of-typerterm(Type, TypeTerm)`
- `regular-type-of-typerterm(Type, type(class, Type, Dim))`
- `typerterm-of-symbol(Symbol, Type)`
- `typerterm-of-expression(Exp, Type)`
- `exists-package(Package)`
- `exists-class(Package, Class)`
- `exists-field(Package, Class, Field)`
- `exists-method(Package, Class, Method)`
- `is-package(Entity)`
- `is-class(Entity)`
- `is-field(Entity)`
- `is-method(Entity)`
- `is-parameter(Entity)`

- `class-cunit(Package, Class, CompilationUnit)`
- `class-reference(Package, Class, Referencing-pef)`
- `field-access(Package, Class, Field, Field-access)`
- `implements-method(Package, Class, Method)`
- `declares-abstract-method(Package, Class, AMethod)`
- `contains-abstract-method(Package, Class, AMethod)`
- `implements-abstract-method(Package, Class, Method, AMethod)`
- `is-method-overriden(SupP, SupC, SupM, SubP, SubC)`
- `call-to-method(Package, Class, Method, Call)`
- `call-to-method-from(SClass-id, SMethod-id, TMethod-id, Call)`
- `called-as-super(TP, TC, TM, Call)`
- `called-as-super-from(SP, SC, TP, TC, Method, Call)`
- `call-to-super(SrcP, SrcC, SrcM, TgtP, TgtC, TgtM, Call)`
- `call-to-super(SrcMethod, TgtMethod, Call)`
- `has-atmost-one-return(Package, Class, Method)`
- `has-no-return(Package, Class, Method)`
- `encl-of-callT(PefId, EnclId, EnclClass)`
- `encl-method-of-callT(PefId, EnclId, EnclClass)`
- `encl-field-of-callT(PefId, EnclId, EnclClass)`
- `encl-class-of-method(Method, Class)`
- `visible-in-class(Field, Package, Class)`
- `visible-in-method(Name, Method)`
- `can-access-class-from-method(Class, Method)`
- `can-access-class(Class1, Class2)`
- `same-package-class(Class1, Class2)`
- `can-access-member(Member, Class)`
- `can-access-method(Method, Class)`
- `can-access-field(Field, Class)`

- `pef-subtree(Pef, SubTreePefs)`
- `pef-children-ids(Pef, Children)`
- `nonprivate-attrs(Package, Class, Fields)`
- `getter-methods(Package, Class, Getters)`
- `setter-methods(Package, Class, Setters)`
- `collect-all-setter-methods(Package, Class, List, Result)`
- `get-parent(Pef, Parent)`
- `get-parent-stmt(Pef, Parent-stmt)`
- `is-stmt(Pef)`
- `get-stmt-container Stmt, Container)`
- `is-multi-stmt-container(Container)`
- `is-single-stmt-container(Container)`
- `get-container-stmts(Container, Stmts)`
- `get-envied-class(Package, Class, Method, EnvPackage, EnvClass)`
- `get-envied-class(MethodID, EnvClassID)`

