



Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR
DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL:

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

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

Dirigida por:
Dra. Yania Crespo González-Carvajal

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.

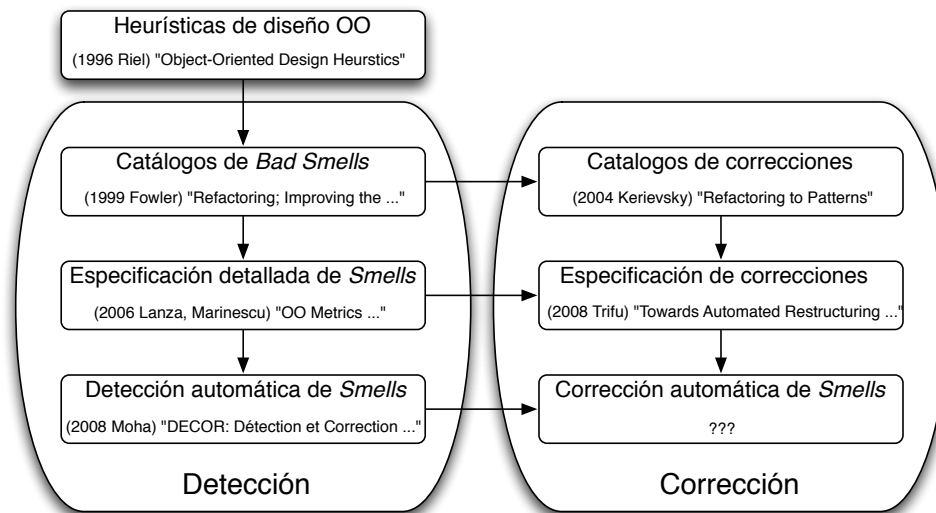


Figura 2.1: Breve resumen histórico sobre la gestión de design smells.

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>

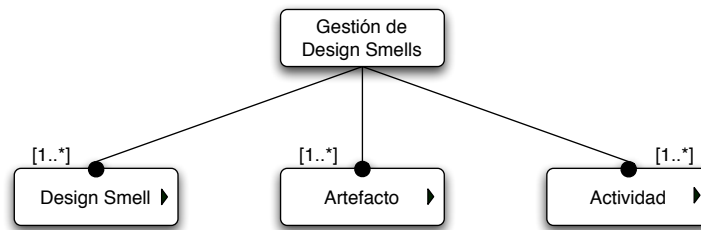


Figura 3.1: Raíz de la taxonomía, puntos de variabilidad principales para comparar los distintos enfoques de design smells.

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

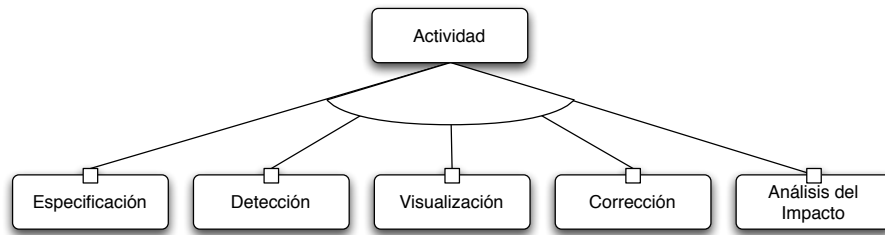


Figura 3.2: Característica actividad y sus subcaracterísticas.

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*.

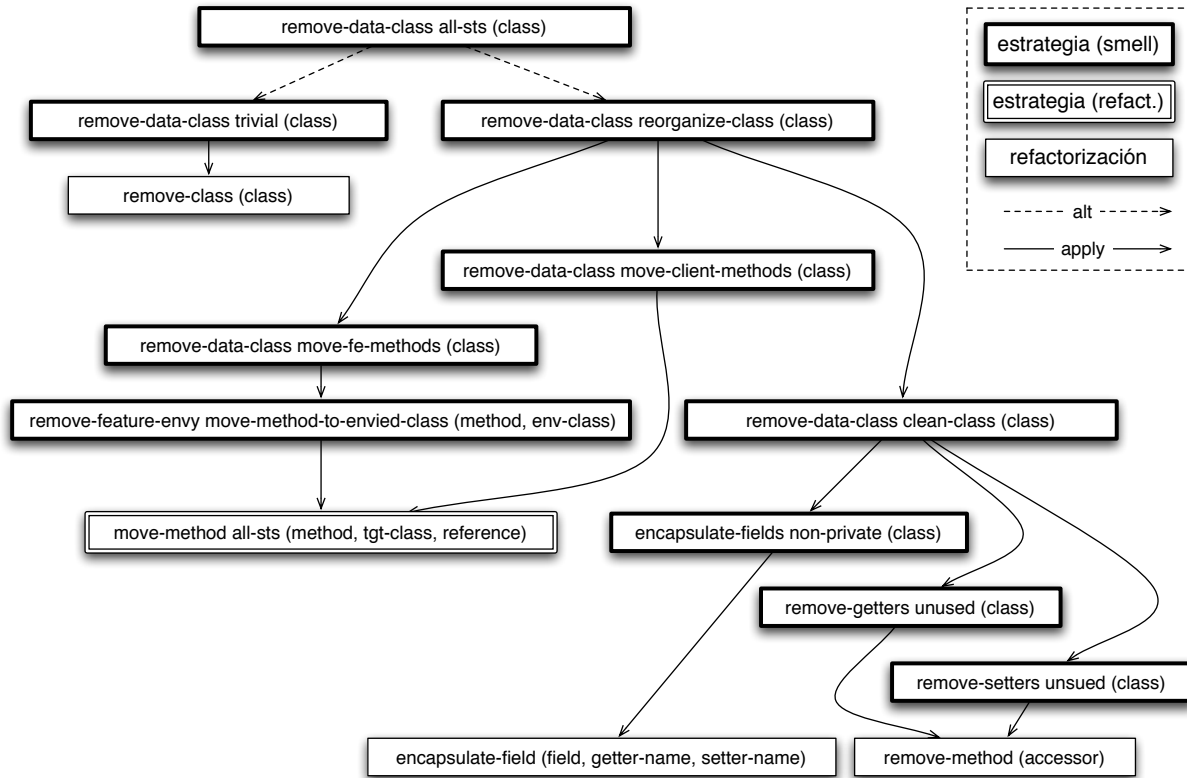


Figura 6.1: Visión general de un conjunto de estrategias sencillas para eliminar una Data Class.

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.

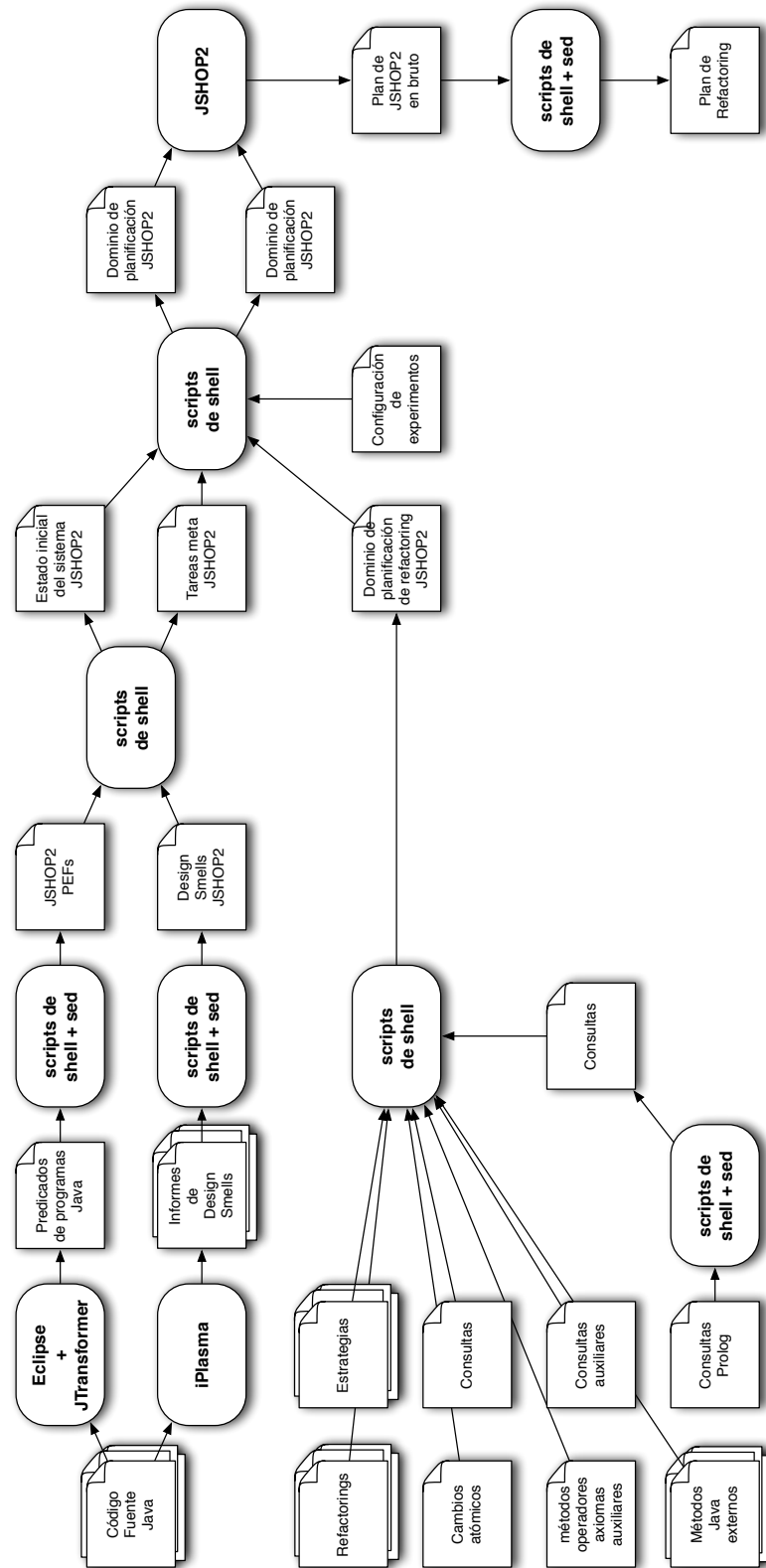


Figura 6.2: *Visión general de las diferentes herramientas utilizadas en los experimentos y de cómo se combinan mediante scripts de SHELL y SED. Las herramientas se muestran como óvalos y los ficheros como hojas con la esquina superior derecha plegada.*

	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 [PVD SM10]. 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.

Bibliografía

- [BF99] 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.
- [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.
- [Bro75] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, Reading, MA , USA, 1975.
- [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.
- [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.
- [Cre00] Yania Crespo. *Incremento del potencial de reutilización del software mediante refactorizaciones*. PhD thesis, Universidad de Valladolid, 2000.
- [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.
- [DDN08] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.
- [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.

- [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.
- [GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning; Theory and Practice*. Morgan Kaufmann, 2004.
- [iPl] iPlasma. LOOSE Research Group;.
- [ISO01] ISO and IEC. ISO/IEC 9126-1:2001, software engineering – product quality, part 1: Quality model, 2001.
- [JSH] JSHOP2. Department of Computer Science, University of Maryland, <http://www.cs.umd.edu/projects/shop>.
- [JTr] JTransformer Engine. ROOTS group, <http://roots.iai.uni-bonn.de/research/jtransformer/>.
- [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.
- [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.
- [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.
- [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.
- [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.
- [Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, October 2002.
- [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.
- [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.
- [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.
- [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.

- [NL06] Colin J. Neill and Phillip A. Laplante. Paying down design debt with strategic refactoring. *IEEE Computer*, 39(12):131–134, 2006.
- [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.
- [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.
- [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.
- [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.
- [RFG05] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving evolvability through refactoring. *SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.
- [Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, April 1996.
- [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.
- [SS07] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring—does it improve software quality? *5th International Workshop on Software Quality*, 0:10, 2007.
- [Tri08] Adrian Trifu. *Towards Automated Restructuring of Object Oriented Systems*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.
- [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.
- [Wak03] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.