

Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones^{*}

Yania Crespo Carlos López, Raúl Marticorena

Dpto. de Informática
Univ. de Valladolid
yania@infor.uva.es

Área de Lenguajes y
Sistemas Informáticos
Univ. de Burgos
{clopezno, rmartico}@ubu.es

Resumen

Uno de los problemas actuales a la hora de refactorizar el código radica en cuándo refactorizar. Hasta el momento, la mayoría de propuestas establecen que el proceso de refactorización nace de la intuición y experiencia del programador.

Partiendo del concepto de “Bad Smell” y a través de métricas, existe la posibilidad de plantear su existencia, no desde un punto de vista subjetivo donde la opinión del programador prima, sino desde un punto objetivo a partir de medidas comparables.

El siguiente trabajo presenta la definición de un soporte al cálculo de métricas para la detección de oportunidades de refactorización. El trabajo se realiza partiendo de una cierta independencia de las métricas respecto del lenguaje concreto, de forma que se puede reutilizar dicho enfoque sobre una familia amplia de lenguajes.

En este punto particular, se soluciona el problema de la pérdida de reutilización en la implementación del cálculo de las métricas mediante una solución basada en frameworks.

Palabras Claves: métricas, bad smells, refactoring, cambio y evolución del software, frameworks.

^{*}Este trabajo ha sido subvencionado por los proyectos : Desarrollo, Integración y Evolución de Sistemas Software, un Enfoque de Líneas de producto. MEC-FEDER, TIN2004-03145. Diagnóstico de Diseños de Sistemas Software y Soporte para su Evolución y Mejora. JCyL VA093/03.

1. Contexto Inicial

Uno de los puntos claves en el proceso de refactorización de código es: ¿cuándo realizar la refactorización? En [6], se propone una lista de pistas o situaciones que sugieren realizar refactoring sobre el código. A dichos síntomas se los denomina “Bad Smells” e inicialmente, su detección se debe realizar *“a partir de la intuición y experiencia del programador”*.

En la actualidad muchos entornos de desarrollo integrados (Eclipse, NetBeans, Visual Studio .NET, Refactoring Browser, etc.) incluyen refactorizaciones. Dichos entornos contienen o permiten incluir plugins para la obtención de métricas.

Sin embargo, a nuestro juicio, existen puntos relacionados entre ambos conceptos sin conectar: pese a disponer de medidas, éstas son utilizadas para la determinación de unos niveles de calidad. No existe una conexión directa entre esas medidas obtenidas, los defectos comunes que pueden ser apuntados (interpretados) por ellas, y por último, las acciones correctoras que pueden ser necesarias para reducir o eliminar dichos defectos.

Actualmente, los entornos de desarrollo integrados, empiezan a sugerir ciertas acciones correctoras a partir de métricas. Tomando como ejemplo el entorno de desarrollo Eclipse, se permite al programador personalizar mensajes de aviso y sugerencias de corrección, para cada una de las mediciones fuera de los límites de las métricas.

Por otro lado, la implementación de las

métricas debe ser realizada para cada entorno/lenguaje orientado a objeto sobre el que se trabaja. Sin embargo una de las propiedades inherentes a las métricas, y en especial en muchas de las orientadas a objetos, es su independencia del lenguaje.

Así pues, a nuestro juicio, las actuales propuestas pueden ser mejoradas. Se apuntan dos mejoras:

- Definir un soporte de métricas independiente del lenguaje. Dicho soporte utilizando frameworks proporciona la posibilidad de reutilización en múltiples entornos o incluso en un mismo entorno con soporte multilenguaje.
- Usar las métricas como indicadores de “Bad Smells” para indicar o sugerir las refactorizaciones adecuadas.

En lo que sigue, este documento se organiza de la siguiente forma: la Sección 2 presenta el estado actual del problema y trabajos relacionados, desde el estudio de métricas y refactoring, hasta aquellos trabajos en los que confluyen ambas líneas. En la Sección 3, se introduce un soporte basado en frameworks para el uso de métricas en el proceso de refactorización. En la Sección 4 se muestra un ejemplo de la aplicación de la relación de las métricas con aquellos síntomas no deseables en el código. Finalmente, en la Sección 5, se presentan las conclusiones y el trabajo futuro.

2. Estado del Arte y Problemas Actuales

Tomando [6] como una de las obras básicas sobre refactorizaciones, se identifican 22 “Bad Smells” definidos desde un punto de vista no formal, con una simple descripción y un conjunto de refactorizaciones sugeridas para su eliminación.

Partiendo de dicho trabajo, en [13] se recogen las opiniones de varios programadores sobre la existencia de “Bad Smells” en un conjunto de códigos, confrontando estos datos, con la información obtenida aplicando métricas sobre los mismos códigos. Propone además una

taxonomía para los “*Bad Smells*”: **Bloaters**, **Object-Oriented Abusers**, **Dispensables**, **Encapsulators**, **Couplers** y **Others**. Para cada uno de los “Bad Smells” se mencionan las métricas que pueden sugerir su existencia, aclarando que dicha asignación se hace desde una perspectiva subjetiva.

Sin embargo este trabajo, en cuanto a relacionar métricas vs. “Bad Smells”, se limita a confrontar los datos obtenidos de encuestas respecto a la correlación con las métricas obtenidas. En particular se limita a tres “Bad Smells” dentro de la categoría **Bloaters**: **Large Class**, **Long Parameter List** y **Duplicated Code**. Mientras que para los dos primeros casos, se pueden utilizar métricas clásicas relativas al tamaño del código [12], en el caso de **Duplicated Code** es necesario utilizar alguna herramienta que detecte código duplicado, puesto que no existen métricas definidas que reflejen esta situación. Este último enfoque basado en otras herramientas, aunque necesario en ciertos casos, se sale del ámbito de estudio de este trabajo.

Sus conclusiones apuntan que no existe una correlación entre la detección de “Bad Smells” basada en la observación humana (subjetiva), frente a la detección basada en medidas obtenidas con el uso de métricas (objetivas). Las encuestas realizadas confirman que la detección de los síntomas de refactorización sin ningún tipo de medida objetiva, llevan a distintos resultados dependiendo del programador (según su experiencia, su vinculación al proyecto, etc.)

Cabe señalar que, como ya se apunta en [18], el autor se concentra precisamente en aquellos “Bad Smell” para los cuales es más fácil aplicar métricas de tamaño de código, con la excepción de **Duplicated Code**. La elección de la herramienta utilizada para la detección de este último “Bad Smell” basada en la repetición de líneas no parece tampoco la más adecuada a la detección de “clones” en el código.

La relación de métricas y refactoring ha sido abordada desde otros puntos de vista. En [3] se definen heurísticas para detectar las refactorizaciones utilizadas para evolucionar software entre distintas versiones. Se utilizan los

valores diferentes de las métricas entre las versiones para deducir qué refactorizaciones han sido aplicadas.

En [17] se propone la detección de refactorizaciones a partir de consultas sobre una base de predicados lógicos. Se definen consultas sobre la información almacenada para sugerir las acciones correctoras a realizar.

Por otra parte en [4, 10] se trabaja con métricas definidas sobre software orientado a objetos, teniendo en cuenta la independencia del lenguaje. Las métricas aplicadas se encuentran limitadas a la información de su metamodelo en el que se carece de información sobre aspectos avanzados de herencia y genericidad. Aunque se menciona el concepto de motor de métricas independiente del metamodelo, no se detalla una descripción de su diseño ni funcionamiento.

Partiendo de todos estos trabajos previos, en el presente artículo se define un framework para el cálculo de métricas sobre un metamodelo, de forma que se reutiliza el enfoque para una familia de lenguajes. Todo esto con el objetivo de poder apoyar la inferencia de qué refactorizaciones aplicar en una aproximación independiente del lenguaje.

3. Framework de Soporte al Cálculo de Métricas

En esta sección se presenta la integración de un soporte al cálculo de métricas sobre soluciones previas, para lograr un proceso de refactorización parcialmente asistido.

Las soluciones actuales para el cálculo de métricas son propuestas respecto a un lenguaje particular. Pese a que es una solución válida, se debe recordar que la mayoría de las métricas, y en mayor medida las orientadas a objetos, toman como punto de partida que son independientes del lenguaje.

Lo que inicialmente parece como una gran ventaja, se demuestra que en la práctica no es aprovechado. Se realiza el mismo esfuerzo de definición e implementación desde cero del cálculo de métricas, para cada entorno y lenguaje particular.

Nuestra solución a este problema se basa en

la existencia de un metamodelo que recoja los elementos básicos en cualquier lenguaje orientado a objeto: clases, atributos, métodos, relaciones de cliente, relaciones de herencia y genericidad. De manera particular sería necesario incluir información sobre las instrucciones de control de flujo, instrucciones de asignación y expresiones, necesarias para el cálculo de métricas como $V(G)$ [14], WMC [2], etc.

Mientras que el metamodelo de UML [15] no contiene información sobre instrucciones, el metamodelo utilizado en nuestro caso sí que mantiene dicha información. Tomando dicho metamodelo ya definido en anteriores trabajos [11], o bien otros metamodelos similares como los de FAMIX [4], la solución propuesta define un framework para el cálculo de métricas con independencia del lenguaje concreto utilizado. Se entiende por framework un conjunto de clases, abstractas y concretas, que definen un comportamiento fácilmente extensible [5].

El objetivo es proponer un framework que pueda utilizarse y extenderse tanto con métricas ya definidas como nuevas métricas, sobre un conjunto amplio de metamodelos. Aunque en particular se valida sobre un metamodelo ya definido, queda abierto a poderse incorporar a otros modelos con pequeñas modificaciones, siguiendo las indicaciones posteriores.

La propuesta se realiza desde el punto de vista del diseño, dejando libertad para su implementación concreta en cualquier lenguaje orientado a objetos.

3.1. Recorrido sobre los Elementos del Metamodelo

Para evitar tener que modificar todas las clases que concuerdan con elementos “medibles”, incluyendo una operación nueva, se aplica el patrón de diseño **Visitor** [7]. La finalidad del patrón es evitar la introducción de operaciones sobre los elementos del modelo, cada vez que se quiere realizar una nueva operación sobre cada uno de ellos. En este caso particular, la necesidad surge al tener que medir propiedades de ciertos elementos del modelo.

Aunque inicialmente pueda parecer un esfuerzo no justificable, introducir dicho patrón favorece la posibilidad futura de poder realizar

otras operaciones sobre dichos elementos. En nuestro caso particular, se ha podido comprobar la utilidad de dicha solución a la hora de diseñar e implementar el subsistema de regeneración de código (con o sin refactorizaciones realizadas sobre los objetos instanciados sobre el metamodelo) empleando de nuevo dicho patrón como pieza clave.

Para ello se introducen operaciones **accept** en cada elemento a visitar. Por otro lado, se define una **interface Visitor** que debe incluir métodos **visit** para cada uno de los elementos sobre los que se quiere realizar una operación (ver Fig. 1).

La dependencia del metamodelo concreto surge en la introducción de las operaciones **accept** en ciertos elementos, y de la dependencia de los elementos concretos a visitar en la definición de la **interface Visitor**.

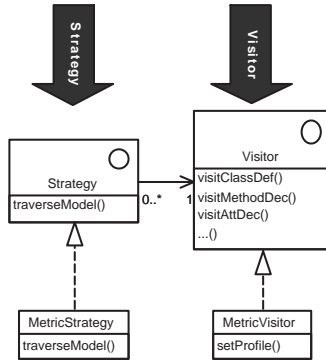


Figura 1: Núcleo del Motor para Obtención de Métricas

El algoritmo de recorrido de los elementos del modelo, se define de manera externa al visitador, permitiendo que mediante un patrón **Strategy** [7], se pueda elegir de manera dinámica el algoritmo particular que conviene en cada caso para acceder a todas las instancias del metamodelo.

3.2. Jerarquía de Métricas a Ejecutar

Las métricas han sido clasificadas de distintas formas. En [9] se recogen distintas clasifica-

ciones más o menos complejas. Particularmente se simplifica centrándonos en los niveles de granularidad (sistema, clase y métodos).

Las métricas que se pueden establecer relativas a los atributos, están ligadas a la clase como contenedora de dichas propiedades. Por ejemplo la métrica NOA (Number Of Attributes) ¹ [12] mide el número de atributos encapsulados en el contexto de una clase.

A la hora de definir las métricas concretas sobre un metamodelo, pueden surgir problemas en cuanto a la pérdida de información que se produce desde el código fuente a las instancias concretas. En nuestro caso la pérdida de información se reduce a las sentencias condicionales, que son almacenadas sin contenido semántico dentro del metamodelo.

Esto impide, por ejemplo, poder definir las métricas relacionadas con la complejidad ciclomática de McCabe [14] de forma independiente. Sin embargo se pueden calcular dentro de las extensiones concretas, con dependencia del lenguaje teniendo en cuenta las palabras claves, y su ejecución se vería soportada por el framework.

La jerarquía de herencia establecida se presenta en la Fig. 2. La clase abstracta **Metric** cumple el rol de **Template Method** [7]. Antes de ejecutarse se realiza un chequeo (método **check**) para comprobar su aplicabilidad. Si es correcto, se ejecuta la medición (método **run**). Mientras que los chequeos se definen en el núcleo del framework, las ejecuciones concretas son parte de la extensión, siguiendo el patrón **Command** [7]. Ambos métodos, **check** y **run** conforman la plantilla de ejecución al invocar al método **calculate**.

Para la recolección de las medidas obtenidas se ha utilizado el patrón **Collecting Parameters** definido en [1]. Se implementa a través de la clase **MetricResult** (ver Fig. 3). Su finalidad es pasar un objeto que recolecta los resultados obtenidos cada vez que se invoca al método **calculate**, sobre un objeto que implemente **IMetric**, similar a la utilización de una pizarra donde se apuntan los resultados colectivos por parte de todos los objetos que colaboran.

¹También conocida como NOF (Number of Fields)

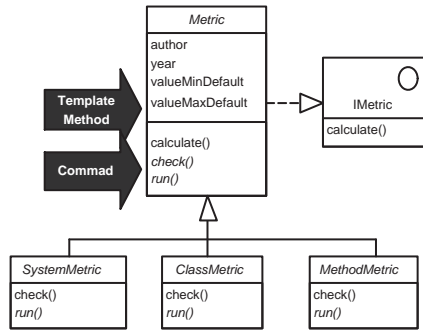


Figura 2: Núcleo de Clases del Framework de Métricas

3.3. Personalización de las Métricas: Perfiles

Las métricas plantean ciertos problemas a la hora de aplicarlas e interpretarlas de manera absoluta. Se puede observar que dependiendo del contexto en que se aplican, los valores límite inferior y superior pueden variar. Por lo tanto el framework debe soportar la personalización de estos valores.

Inicialmente se instancia cada una de las métricas con los valores por defecto recomendados. Estos valores son subjetivos y deben ser ajustados mediante observación empírica, afinando y ajustando dichos valores a través de la experiencia.

Para ello se crea una clase de envoltura, **MetricConfiguration** (ver Fig. 3), que permite ajustar la definición inicial de una métrica, pudiendo sobrescribir los valores por defecto. Esto permite ajustar las métricas al contexto particular donde se aplican.

Mediante la agrupación de distintas configuraciones en la clase **MetricProfile** (ver Fig. 3), se posibilita la definición de distintos perfiles por parte del programador. Como se ha observado, dependiendo del dominio en que se desarrolle los valores cambiarán. No se abordan, por el momento, aspectos relativos a la persistencia y recuperación de los perfiles.

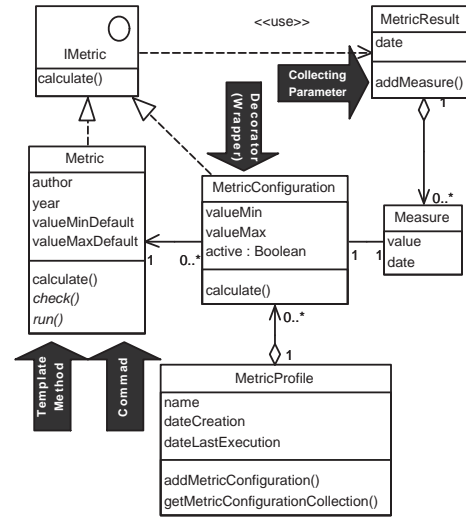


Figura 3: Personalización del Cálculo con Perfiles

3.4. Cálculo de las Medidas

Colocando todas las piezas juntas, el proceso concreto de recorrido y cálculo se inicia por medio de una estrategia para métricas, que se apoya sobre un visitador concreto para cada uno de los elementos del metamodelo a visitar. El visitador lleva asociado un cierto perfil concreto de métricas.

Sobre cada elemento se aplican aquellas métricas configuradas en el perfil que se está aplicando. Los resultados obtenidos para cada objeto instanciado sobre el metamodelo se recoge como una medida que permite trazabilidad hacia la métrica concreta con la que se ha obtenido a través de la clase **MetricConfiguration**, y el objeto sobre el que ha sido calculada. Las medidas se agrupan en lo que se denomina el “resultado de métricas” (clase **MetricResult**) para posibilitar el análisis y presentación de resultados posteriores.

3.5. Ejemplo de Validación del Framework

El framework ha sido implementado, sobre un metamodelo ya definido. Dicho metamodelo

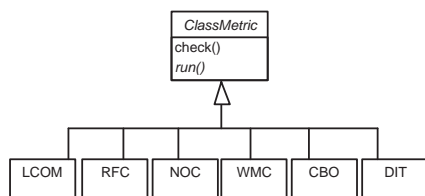


Figura 4: Ejemplo de Extensión Concreta al Framework

soporta, entre otros, los conceptos de clases y herencia. Sobre dicho framework de métricas se ha abordado la implementación de algunas métricas como DIT (Depth Inheritance Tree) o NOC (Number Of Children) [8] entre otras.

Ambas métricas son definidas a nivel de clase. Por lo tanto, una vez determinado su contexto, se definen por extensión de **ClassMetric** (ver Fig. 4). El cuerpo del método **run** se redefina para obtener un valor numérico correspondiente a partir de la información extraída del modelo y representada como instancia de **Measure**. El punto de entrada para el cálculo de la métrica aplicada es en este caso una clase del código analizado. A partir de dicha clase se navega a través de sus relaciones de herencia para obtener su profundidad y el número de hijos.

Para su ejecución sobre un código concreto el esfuerzo recae en el framework. El programador simplemente debe incluir dichas métricas en el perfil concreto a aplicar. La implementación tanto del metamodelo como del framework de métricas se ha realizado sobre Java, pero dado el carácter abierto del diseño no hay impedimentos para implementarlo con cualquier otro lenguaje orientado a objetos.

3.6. Puntos Fuertes y Débiles de la Propuesta

El principal beneficio de este planteamiento es que una vez implementado el framework, se tiene una herramienta de obtención de métricas para un conjunto de lenguajes orientados a objetos muy amplio, tantos como parsers hacia el metamodelo se construyan. En nuestro

trabajo, tenemos implementado un metamodelo (MOON) y contamos con parsers hacia el metamodelo para Java y Eiffel.

Dentro de las posibles mejoras al framework, se apunta:

- la introducción del patrón **Observer** [7] para actualizar y recalculas sólo las métricas asociadas a los elementos modificados.
- la introducción de filtros adicionales para indicar que ciertas métricas no se aplican por igual a un cierto tipo de elemento. Por ejemplo, en los constructores de clases dentro de aquellos lenguajes que lo permiten, la métrica NOP (Number Of Parameters) [16] podría ser relajada en sus valores máximos.
- incorporar una capa de presentación gráfica para la asistencia a la interpretación de valores obtenidos.

4. Relación entre Bad Smells y Métricas

Partiendo del trabajo de [13] donde la validación de la relación entre ciertas métricas y “Bad Smells” queda pendiente, hemos planteado un caso de estudio particular para mostrar la utilidad de un enfoque de detección de “Bad Smells” basado en métricas. De esta forma también se continúa con la validación del framework de métricas construido.

Suponiendo que se debe refactorizar un proyecto de gran tamaño del que se dispone del código fuente, pero no experiencia sobre el dominio surge una cuestión: ¿por dónde empezamos a refactorizar? A continuación se presenta un caso de estudio con un proyecto de código libre en el que sin mayor conocimiento del mismo, se sugieren oportunidades de refactorización.

Como ejemplo se ha tomado un producto de código abierto JFreeChart (www.jfree.org). Se trata de una biblioteca de clases Java para elaborar distintos tipos de gráficos. Las mediciones se realizan sobre la última versión (1.0.0-pre2).

El ámbito de estudio se centra en aquellos “Bad Smells” que pueden ser determinados con métricas ampliamente aceptadas e independientes del lenguaje. Fuera de este ámbito quedan cuestiones como convenciones de nombres, código duplicado, etc.

En particular se toman dentro de la categoría **Dispensables** dos “Bad Smells” con una fuerte correlación [13]: **Data Class** y **Lazy Class**. El objetivo es determinar qué clases son sospechas en dicha biblioteca de presentar estos síntomas.

El ejemplo se centra en métricas aplicadas a nivel de clase. Se han seleccionado métricas de tamaño como NOA, NOM, etc. [12] y orientadas a objetos, como las definidas en [2]: WMC, NOC, DIT, LCOM, CBO y RFC. Dentro de este conjunto, se han seleccionado además métricas con fuerte correlación como NOA, NOM, WMC y LCOM o correlación negativa como DIT. Dichas correlaciones se han validado sobre la propia biblioteca de clases.

4.1. Bad Smell: Data Class

Se caracteriza por clases con un alto número de atributos (NOA) y gran número de métodos (NOM) de acceso/mutadores. La complejidad de la clase es pequeña (WMC) y suelen estar cohesionadas (LCOM).

Con este filtro, tomando las cinco clases cuyas métricas tienen valores más próximos se han localizado: **AbstractRenderer**, **ChartPanel**, **PiePlot**, **XYPlot** y **CategoryPlot**. Si se observa su código se pueden distinguir que todas ellas son clases con gran número de métodos **get** y **set**, y con poca funcionalidad.

Refactorizaciones a aplicar [6]: **Move Method** para incorporar más funcionalidad a dichas clases.

4.2. Bad Smell: Lazy Class

Se caracteriza por clases con un bajo número de atributos (NOA) y de métodos (NOM). La complejidad de la clase es pequeña (WMC). Su valor de profundidad en la jerarquía de herencia (DIT) es bajo y sin hijos (NOC). Por lo tanto no aporta funcionalidad, ni directa ni indirectamente por herencia. Su cohesión

(LCOM) entre métodos también suele ser baja.

Con este filtro, se comprueba que:

- si se fija un valor DIT=1, se detectan aquellas clases meramente funcionales, sin estado, que además no realizan mucho trabajo.
- para valores más altos de DIT se localizan clases que no incorporan ninguna funcionalidad. Las nombres de dichas clases comienzan por **Default**, lo que indica un comportamiento por defecto.
- muchas de las clases localizadas con este filtro implementan el patrón **Factory Method** [7].

Refactorizaciones a aplicar [6]: **Move Method**, **RemoveClass**, **CollapseHierarchy**, **Inline Class**.

5. Conclusiones y Líneas de Trabajo Futuro

La aplicación de las métricas y su cálculo a través de herramientas, evita una inspección manual del código, pudiendo localizar con menor esfuerzo aquellas clases propensas a ser refactorizadas.

La posibilidad de dar soporte al cálculo de métricas para la inferencia de refactorizaciones, con un objetivo de independencia del lenguaje, ofrece la posibilidad de reutilizar la relación entre “Bad Smells”, métricas y por lo tanto, de las refactorizaciones a aplicar. El objetivo final es su integración en varios entornos, o en entornos multilingaje cada vez más presentes.

Aunque conscientes de la gran cantidad de trabajo abierto, se apuntan beneficios en este tipo de aproximaciones, desde el punto de vista de la reutilización de los esfuerzos realizados en la definición del soporte a métricas y refactoring.

Dentro de las líneas de trabajo abiertas en base al actual trabajo se encuentran:

- Afrontar los problemas a la hora de implementar métricas que necesiten de aspectos concretos de los lenguajes como puede

ser inspeccionar las sentencias condicionales y expresiones contenidas.

- Continuar con la validación empírica del framework de métricas para la detección de “Bad Smells”, y por lo tanto, la inferencia de refactorizaciones a aplicar.

Referencias

- [1] Kent Beck. *Smalltalk: best practice patterns*. Prentice-Hall, Inc., 1997.
- [2] Shyam R. Chimdaber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions On Software Engineering*, 20:476–493, 1994.
- [3] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA’2000*, pages 166–177. ACM Press, 2000.
- [4] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, Institute of Computer Science and Applied Mathematic. University of Bern, 1999.
- [5] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph Johnson. *Building Applications Frameworks. Object-Oriented Foundations of Framework Design*. Wiley, 1999.
- [6] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison Wesley, 2000.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [8] Martin Hitz and Behzad Montazeri. Chidamber and kemerer’s metrics suite: A measurement theory perspective. *Software Engineering*, 22(4):267–271, 1996.
- [9] David A. Lamb and Joe R. Abouander. Data model for object-oriented design metrics. Technical report, Department of Computing and Information Science. Queens’s University., 1997.
- [10] Michele Lanza and Stéphane Ducasse. Beyond language independent object-oriented metrics: Model independent metrics. In *QA00SE 2002*, pages 77–84, 2002.
- [11] Carlos López, Raúl Marticorena, and Yania Crespo. Hacia una solución basada en frameworks para la definición de refactorizaciones con independencia del lenguaje. In Jaime Gómez Ernesto Pimentel, Nieves R. Brisaboa, editor, *Actas JISBD’03, Alicante, Spain ISBN: 84-688-3836-5*, pages 251–262, November 2003.
- [12] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
- [13] Mika Mäntylä. Developing new approaches for software design quality improvement based on subjective evaluations. In *ICSE*, pages 48–50. IEEE Computer Society, 2004.
- [14] Tomas McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
- [15] OMG. Unified modeling language: Superstructure version 2.0. <http://www.uml.org>, 2004.
- [16] Meilir Page-Jones. *The practical guide to structured systems design: 2nd edition*. Yourdon Press, Upper Saddle River, NJ, USA, 1988.
- [17] Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proc. 7th European Conf. on Software Maintenance and Reengineering*, pages 91 – 100, Benvento, Italy, 2003. IEEE Computer Society.
- [18] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003.