

Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR
DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL:

Detección de defectos de diseño mediante métricas de código

Presentada por Carlos López Nozal para optar
al grado de doctor por la Universidad de Valladolid

Dirigida por:
Dra. Yania Crespo González-Carvajal
Dra. María Esperanza Manso Martínez

Octubre, 2012

Con todo mi *corazón* para Ana y nuestros hijos Nicolás y Lucas

AGRADECIMIENTOS

A mis directoras de tesis Dr. Yania Crespo González-Carvajal y Dr. María Esperanza Manso Martínez por su dedicación y todo lo que he aprendido con ellas. Sin sus conocimientos y sus consejos este trabajo nunca hubiera existido.

A mis compañeros del grupo de investigación de la Universidad de Valladolid GIRO, por su gran ayuda y por facilitarme todo el soporte necesario de investigación. Su apoyo y su acompañamiento han suavizado la dureza del camino.

A mis compañeros del Área de Lenguajes y Sistemas Informáticos de la Universidad de Burgos, por animarme a hacer esta tesis y por ayudarme siempre en la medida de sus posibilidades.

A todos los que en algún momento he aburrido con mis “defectos de diseño”. Especialmente a todos los participantes en la encuesta: profesionales en el desarrollo de aplicaciones software, profesores y alumnos de Ingeniería Informática de la Universidad de Burgos.

ÍNDICE GENERAL

Índice de figuras	XI
Índice de tablas	XIII
1. Introducción	1
1.1. Defecto de diseño	1
1.2. Calidad y métricas en ingeniería del software	3
1.3. Experimentación en ingeniería del software	5
1.4. Definición del contexto	8
1.5. Objetivos de investigación	8
1.6. Estructura de la tesis	10
1.6.1. Capítulo 2: Revisión de defectos de diseño y su gestión	10
1.6.2. Capítulo 3: Proceso de medición modificado y su validación	10
1.6.3. Capítulo 4: Proceso dinámico de gestión de defectos	12
1.6.4. Capítulo 5: Framework para la medición y gestión de defectos	12
2. Revisión de defectos de diseño y su gestión	15
2.1. Catálogos de defectos de diseño	16
2.2. Visión general del estudio	17
2.2.1. Notación del modelo de características	18
2.2.2. Características del nivel superior de la gestión de defectos de diseño	20
2.3. Defectos de diseño	21
2.3.1. Tipo de defecto	21
2.3.2. Nivel del defecto	23
2.3.3. Ámbito del defecto	23
2.3.4. Propiedades del defecto	23
2.4. Artefacto	24
2.4.1. Tipo de artefacto	24
2.4.2. Versiones	25
2.4.3. Tipo de representación	26
2.5. Actividad	27
2.5.1. Especificación	28
2.5.2. Detección	29

2.5.3.	Visualización	31
2.5.4.	Corrección	32
2.5.5.	Análisis de impacto	33
2.6.	Caracterización de herramientas: inCode y JDeodorant	34
2.7.	Caracterización de la tesis	37
3.	Proceso de medición modificado y su validación	41
3.1.	El proceso de medición	42
3.1.1.	Estudio de los valores umbrales en distintos productos	42
3.1.2.	Estudio de los valores umbrales en la evolución del producto	45
3.2.	Propuesta de un proceso de medición	48
3.3.	Estudio empírico para validar el proceso de medición	49
3.4.	Caso de estudio 1: Medidas en Plugings de Eclipse	50
3.4.1.	Definición del caso de estudio: objetivos, hipótesis y variables	51
3.4.2.	Planificación	51
3.4.2.1.	Selección del contexto y sujetos	51
3.4.2.2.	Objetos del estudio	52
3.4.2.3.	Diseño del caso de estudio	52
3.4.2.4.	Instrumentación	52
3.4.3.	Operación	54
3.4.4.	Análisis	56
3.4.4.1.	Resultados observados	57
3.4.4.2.	Valores umbrales dependientes de la naturaleza de la entidad	58
3.4.5.	Validación	59
3.5.	Réplica del caso de estudio 1: Mediciones en trabajos fin de estudios	60
3.5.1.	Definición de la réplica del caso de estudio 1	61
3.5.2.	Planificación	61
3.5.2.1.	Selección del contexto	61
3.5.2.2.	Selección de objetos	62
3.5.2.3.	Instrumentación	63
3.5.3.	Operación	64
3.5.4.	Análisis	65
3.5.4.1.	Resultados observados	65
3.5.4.2.	Valores umbrales dependientes de la naturaleza de la entidad	66
3.6.	Caso de estudio 2: Comparación de métodos de clasificación de entidades	68
3.6.1.	Definición del caso de estudio: objetivo, hipótesis y variables	68
3.6.2.	Planificación	69
3.6.2.1.	Selección de sujetos y objetos	69
3.6.2.2.	Diseño de la encuesta	70
3.6.2.3.	Instrumentación	71
3.6.3.	Operación	71
3.6.4.	Análisis	72
3.6.4.1.	Datos observados	75
3.6.4.2.	Análisis descriptivo	77

3.6.4.3.	Estudio de las hipótesis sobre concordancia en las clasificaciones	81
3.6.4.4.	Amenazas a la validez	92
3.6.5.	Conclusiones sobre la clasificación automática	92
3.7.	Conclusiones de la experimentación	94
4.	Proceso dinámico de gestión de defectos	97
4.1.	Proceso genérico de gestión de defectos de diseño	98
4.1.1.	Participantes	98
4.1.2.	Tareas	99
4.1.3.	Productos	99
4.2.	Aplicación del proceso	100
4.2.1.	Datos de entrada	100
4.2.2.	Validación de las decisiones del analista	101
4.2.3.	Plantilla de descripción del defecto	103
4.3.	Aplicación del proceso a <i>God Class</i>	103
4.3.1.	Clasificación y motivación	103
4.3.2.	Heurísticas para su detección	103
4.3.3.	Definición	105
4.3.4.	Experimentación	105
4.3.5.	Validación	107
4.4.	Aplicación del proceso a <i>Data Class</i>	108
4.4.1.	Clasificación y motivación	108
4.4.2.	Heurísticas para su detección	108
4.4.3.	Definición	109
4.4.4.	Experimentación	109
4.4.5.	Validación	111
4.5.	Aplicación del proceso a <i>Feature Envy</i>	112
4.5.1.	Clasificación y motivación	112
4.5.2.	Heurísticas para su detección	112
4.5.3.	Definición	113
4.5.4.	Experimentación	113
4.5.5.	Validación	117
4.6.	Estudio empírico para evaluar la eficiencia en métodos que identifican defectos	117
4.7.	Evaluación de la eficiencia de métodos de identificación del defecto <i>God Class</i>	118
4.7.1.	Definición del caso de estudio: hipótesis y variables	118
4.7.2.	Planificación	119
4.7.2.1.	Diseño y variables de estudio	119
4.7.2.2.	Selección de sujetos y objetos	122
4.7.2.3.	Instrumentación	122
4.7.3.	Operación	123
4.7.4.	Análisis	124
4.7.4.1.	Estudio de la eficiencia de los clasificadores para identificar <i>God Class</i>	124

4.7.4.2.	Comparación de los métodos de clasificación	129
4.7.4.3.	Amenazas a la validez	129
4.8.	Evaluación de la eficiencia de métodos de identificación del defecto <i>Data Class</i>	130
4.8.1.	Análisis	130
4.8.1.1.	Estudio de la eficiencia de los clasificadores para identificar <i>Data Class</i>	131
4.9.	Evaluación de la eficiencia de métodos de identificación del defecto <i>Feature Envy</i>	135
4.9.1.	Análisis	136
4.9.1.1.	Estudio de la eficiencia de los clasificadores para identificar <i>Feature Envy</i>	137
4.10.	Conclusiones de la experimentación	140
5.	Framework para la medición y la gestión de defectos	143
5.1.	Framework de soporte para el cálculo de medidas	144
5.1.1.	Recorrido sobre los elementos del metamodelo	144
5.1.2.	Jerarquía de métricas	145
5.1.3.	Personalización de las métricas: Perfiles	146
5.1.4.	Cálculo de las medidas	147
5.1.5.	Ejemplo de validación del framework	148
5.1.6.	Puntos fuertes y débiles de la propuesta	148
5.2.	Prototipo de herramienta adaptada al proceso de medición propuesto . .	149
5.2.1.	Adaptación de la herramienta RefactorIt	149
5.2.1.1.	Clasificación de entidades de código	151
5.2.1.2.	Clasificación de entidades en las categorías consideradas	151
5.3.	Framework de soporte para la gestión de defectos	154
5.3.1.	Definición de defectos de diseño	154
5.3.2.	Integración con el framework de métricas	155
5.3.3.	Puntos fuertes y débiles de la propuesta	156
5.4.	Prototipos de herramientas adaptados al proceso de gestión de defectos	157
5.4.1.	Defectos arquitectónicos	157
5.4.2.	Gestión integral de defectos de diseño	159
5.4.3.	Conclusiones sobre el prototipado de gestión de defectos	161
6.	Conclusiones y Líneas de Trabajo Futuras	165
6.1.	Conclusiones	165
6.2.	Líneas de Trabajo Futuras	168
	Apéndices	171
A.	Encuesta de clasificación de entidades de código	173
A.1.	Descripción	173
A.2.	Perfil del inspector de código	173
A.3.	Estereotipos UML	174
A.4.	Clasificación de entidades de código de la aplicación Visor de Imágenes .	176
A.5.	Clasificación de entidades de código EclEmma a partir de código fuente	178

B. Publicaciones	181
B.1. Congresos, talleres y revistas	181
B.2. Publicaciones y su relación con capítulos	183
Referencias	187

ÍNDICE DE FIGURAS

1.1. Resumen de la experimentación realizada en el Capítulo 3	11
1.2. Resumen de la experimentación realizada en el Capítulo 4	12
2.1. Notación del diagrama de características utilizado en el estudio.	19
2.2. Nivel superior del modelo de características para comparar soluciones en la gestión de defectos de diseño	20
2.3. Característica defecto de diseño y sus subcaracterísticas	21
2.4. Característica artefacto objetivo y sus características	24
2.5. Característica actividad y sus características	27
2.6. Grado de automatización soportada en una actividad de gestión de defectos de diseño	28
2.7. Resumen de la situación actual de la actividad de especificación.	29
2.8. Resumen de la actual situación de la actividad de detección	30
2.9. Resumen de la situación actual de la actividad de visualización	32
2.10. Resumen de la situación actual de la actividad de corrección	33
2.11. Resumen de la situación actual de análisis de impacto	35
3.1. Proceso de medición definido por Sommerville	42
3.2. Resultados descriptivos globales	44
3.3. Gráficos de cajas de WMC de diferentes productos y tipos de software	45
3.4. Análisis descriptivo de JFreeChart	46
3.5. Análisis descriptivo de JHotDraw	47
3.6. Análisis descriptivo de JUnit	47
3.7. Proceso de medición propuesto	49
3.8. Evaluación de proyectos	62
3.9. Intervalos de confianza de Tukey para las diferencias del porcentaje de documentación	67
4.1. Proceso dinámico de gestión de defectos	98
4.2. Contexto de aplicación del proceso de gestión de defectos	101
5.1. Núcleo del motor para la obtención de métricas	145
5.2. Núcleo de clases del framework de métricas	146

5.3. Personalización del cálculo con perfiles	147
5.4. Ejemplo de extensión concreta del framework	148
5.5. Proceso de medición con la herramienta RefactorIt	150
5.6. Definición de valores umbrales para cada métrica y categoría con RefactorItUBU	152
5.7. Inspección de entidades de código con RefactorItUBU	152
5.8. Núcleo de la gestión de defecto de diseño	154
5.9. Relación defectos de diseño con valores de medidas	156
5.10. Perspectivas del plugin ADF	158
5.11. Interfaz gráfica del prototipo para seleccionar el profile en la tarea de identificación de defectos de diseño	161
5.12. Interfaz gráfica del prototipo con las tablas de entidades-defectos inicial y revisada	162
5.13. Interfaz gráfica del prototipo para la tarea de definición de defectos de diseño	162
5.14. Interfaz gráfica del prototipo para la tarea de experimentación: generar nuevos clasificadores	163
5.15. Interfaz gráfica del prototipo para la tarea de experimentación: configurar clasificación de la naturaleza de la entidad basada en nombres	163
5.16. Interfaz gráfica del prototipo para la tarea de validación de los clasificadores	164
A.1. Interfaz gráfica visor de imágenes	176
A.2. Estructura y nombres de las entidades de código	177

ÍNDICE DE TABLAS

1.1. Tabla cruzada de capítulos y objetivos	10
2.1. Resumen de los trabajos de defectos	18
2.2. Característica defecto de diseño en Incode y JDeodorant	36
2.3. Característica artefacto en inCode y JDeodorant	36
2.4. Característica actividad en inCode y JDeodorant	37
2.5. Característica defecto de diseño en esta tesis	38
2.6. Característica artefacto en esta tesis	39
2.7. Característica actividad en esta tesis	39
3.1. Caracterización de los casos de estudio	50
3.2. Conjunto de métricas definido en RefactorIt	53
3.3. Información de los proyectos elegidos	54
3.4. Convención de nombres en inglés para clasificar entidades	56
3.5. p-valores observados en el test de K-W, subrayados los no significativos al nivel 0.05	58
3.6. Resumen de resultados significativos al nivel 0.05	59
3.7. Métricas de paquetes: Valores umbrales recomendados según la naturaleza del problema	60
3.8. Métricas de clases: Valores umbrales recomendados según la naturaleza del problema	60
3.9. Métricas de métodos: Valores umbrales recomendados según la naturaleza del problema	61
3.10. Herramientas de medida de código	63
3.11. Conjunto de métricas definido en SourceMonitor	64
3.12. Resultado de la medición de un proyecto	65
3.13. Tamaño de la muestra, según la naturaleza del problema	65
3.14. p-valores observados, subrayados los no significativos al nivel 0.05	66
3.15. Métricas de subsistemas: Valores umbrales recomendados según la naturaleza del problema	68
3.16. Esquema de la encuesta	71
3.17. Descripción de los casos de estudio	72
3.18. Preguntas e hipótesis	72

3.19. Interpretación de valores de Kappa	75
3.20. Resumen de los perfiles de los sujetos en DMSII y GIRO	76
3.21. Resultados detallados del caso de estudio DMSII	77
3.22. Resultados detallados de la réplica GIRO	78
3.23. Entidades propuestas en la réplica GIRO	79
3.24. Resumen del caso de estudio DMSII	80
3.25. Resumen de la réplica GIRO	81
3.26. Resumen del caso de estudio DMSII y la réplica GIRO	82
3.27. Concordancia en la definición de conceptos	83
3.28. Concordancia en la clasificación de entidades	86
3.29. Resumen de resultados significativos para la concordancia en clasificar, por estereotipos, objetos y experiencia $H_{0,2a}$	88
3.30. Resultados de concordancia de evaluación humana vs. automática res- pecto a entidades acotadas	90
3.31. Resumen del grado de concordancia entre la evaluación humana y la automática, respecto a entidades no acotadas, por expertos de GIRO	92
3.32. Resumen de las conclusiones de los casos de estudio	95
4.1. Salidas del proceso de aprendizaje en una predicción binaria	103
4.2. Descripción de la aplicación del proceso definido en 4.1	104
4.3. Conjunto de datos parciales de <i>God Class</i>	106
4.4. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación EclEmma 2.1.0	106
4.5. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación JFreeChart 1.0.14	107
4.6. Evaluación de los clasificadores para el defecto <i>God Class</i>	107
4.7. Conjunto de datos parciales de <i>Data Class</i>	110
4.8. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación EclEmma 2.1.0	110
4.9. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación JFreeChart 1.0.14	111
4.10. Evaluación de los clasificadores para el defecto <i>Data Class</i>	111
4.11. Conjunto de datos parciales de <i>Feature Envy</i>	113
4.12. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación JFreeChart 1.0.14	113
4.13. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación EclEmma 2.1.0	116
4.14. Evaluación de los clasificadores para el defecto <i>Feature Envy</i>	117
4.15. Conjunto parcial de métricas de código definido en RefactorIt	120
4.16. Tabla de costes	121
4.17. Diseño del caso de estudio	121
4.18. Defectos de diseño identificados por Incode 2.07 y JDeodorant 4.0.12	123
4.19. Preguntas e hipótesis	124
4.20. Evaluación de los clasificadores para el defecto <i>God Class</i> sobre el con- junto de datos generado a partir de JFreeChart 1.0.14	125

4.21. Evaluación de los clasificadores para el defecto <i>God Class</i> sobre el conjunto de datos generado a partir de EclEmma 2.1.0	126
4.22. Resumen del análisis del caso de estudio <i>God Class</i> respecto $H_{0,1a}$ y $H_{0,1b}$	128
4.23. Predicciones del defecto <i>God Class</i> con JDeodorant e Incode	129
4.24. Análisis de la concordancia entre los métodos Incode y JDeodorant . . .	129
4.25. Preguntas e hipótesis	131
4.26. Evaluación de los clasificadores para el defecto <i>Data Class</i> sobre el conjunto de datos generado a partir de JFreeChart 1.0.14	132
4.27. Evaluación de los clasificadores para el defecto <i>Data Class</i> sobre el conjunto de datos generado a partir de EclEmma 2.1.0	133
4.28. Resumen del análisis del caso de estudio <i>Data Class</i> respecto $H_{0,1a}$ y $H_{0,1b}$	135
4.29. Preguntas e hipótesis	136
4.30. Evaluación de los clasificadores para el defecto <i>FeatureEnvy</i> sobre el conjunto de datos generado a partir de JFreeChart 1.0.14	137
4.31. Evaluación de los clasificadores para el defecto <i>Feature Envy</i> sobre el conjunto de datos generado a partir de EclEmma 2.1.0	138
4.32. Resumen del análisis del caso de estudio <i>Feature Envy</i> respecto $H_{0,1a}$ y $H_{0,1b}$	140
4.33. Resumen de los resultados del estudio empírico	141
B.1. Referencias cruzadas entre publicaciones y capítulos	185

Capítulo 1

INTRODUCCIÓN

En este capítulo se presentan los conceptos principales y la terminología relacionados con esta tesis. En concreto se describirán brevemente los siguientes conceptos: defectos de diseño, calidad y métricas del software y experimentación en ingeniería del software. A partir de ellos se contextualiza el trabajo de investigación que hemos llevado a cabo. Por último presentaremos los objetivos de esta tesis y su estructura.

1.1. Defecto de diseño

Un *defecto de diseño* describe una situación que sugiere un problema potencial en la estructura del software. Para decidir si el problema es real o relevante, la situación tiene que ser examinada con más detalle en su contexto particular. En la literatura este concepto tiene un plétora de términos diferentes para referirse a una familia de conceptos similares:

- **code smells** en [FBB⁺99] y <http://c2.com/cgi/wiki?CodeSmell>
- **bad smells** en [FBB⁺99]
- **disharmonies** en [LM06]
- **design flaws** en [SLT06, Tri08]
- **defects** en [MGL⁺09, TSFB99]
- **antipatterns** en [BMMM98, page 159]
- **design smells** en [Mar03]

El término “code smell” fue introducido por Kent Beck para definir problemas estructurales en el código fuente que pueden ser detectados por desarrolladores experimentados:

“Un code smell es una pista para indicar que algo puede funcionar erróneamente en algún lugar del código.”

La estructura de código sospechosa puede que directamente no provoque daños (en términos de fallos y errores de ejecución), pero tiene un impacto negativo en la estructura global del sistema y, como consecuencia, sobre sus factores de calidad. La presencia de “code smells” es un indicador del desorden en el diseño de un sistema, dificultando su comprensión y mantenimiento. Además, su presencia puede avisar de problemas de desarrollo como errores en la elección de la arquitectura o incluso malas prácticas de gestión. El término “code smell” se hizo popular en el libro de Fowler [FBB⁺99], donde se encuentra un conjunto de “bad smells in code”. Un buen ejemplo de “code smell” es la presencia de código duplicado. Esta situación aparece cuando un conjunto de instrucciones, una expresión, un algoritmo etc. es utilizado en muchos lugares del sistema. Este “code smell” hace que la estructura del sistema sea más difícil de mantener, convirtiéndose en un foco de futuros fallos de ejecución en el caso de tener que cambiar una copia del código duplicado.

Los otros términos referenciados anteriormente, “disharmonies”, “design flaws”, “defects”, “antipatterns”, son conceptos similares a “code smells” pero más generales. Los términos cubren un rango completo de problemas relacionados con la estructura del software. Los diferentes defectos afectan al software en distintos niveles de granularidad, desde métodos (Ej. Lista larga de parámetros [FBB⁺99, page 78]), hasta la arquitectura completa del sistema (Ej. Stovepipe System [BMMM98, page 159]). Es destacable mencionar la diferencia encontrada en la literatura entre los términos “design flaws” y “flaws” ya que el último está más frecuentemente asociado a errores en tiempo de ejecución.

Robert C. Martin [Mar03] se refiere a “design smells” como defectos de alto nivel que causan la decadencia de la estructura del sistema software, que puede ser detectada cuando el software empieza a mostrar los siguientes problemas:

- **Rigidez:** El diseño es difícil de cambiar porque cada cambio implica otros muchos cambios en otras partes del sistema.
- **Fragilidad:** El diseño es fácil de romper. Los cambios causan al sistema roturas en lugares que no tienen relación conceptual con la parte que fue cambiada.
- **Inmovilidad:** Es difícil descomponer el sistema en componentes que puedan ser utilizados en otros sistemas.
- **Viscosidad:** Hacer las cosas bien es más difícil que hacerlas mal. Es más fácil utilizar atajos rápidos.
- **Complejidad innecesaria:** El sistema está supra-diseñado, contiene infraestructuras que no añaden beneficio directo.
- **Repetición innecesaria:** El sistema contiene estructuras repetitivas que podrían ser unificadas en una única abstracción.
- **Opacidad:** El sistema es difícil de leer y comprender y no expresa correctamente su objetivo.

A pesar de la terminología diferente utilizada en la literatura, todos los términos mencionados tienen en común que describen problemas relacionados con malos diseños.

Para simplificar y unificar la terminología existente, en esta tesis se utilizará el término *defecto de diseño* para representar:

“Un problema encontrado en la estructura software (en artefactos de código o de diseño), que no produce errores ni de compilación ni de ejecución, pero afecta negativamente a factores de calidad del software. Este efecto negativo podría ocasionar errores en el futuro.”

1.2. Calidad y métricas en ingeniería del software

La necesidad de medir en la ingeniería del software se podría resumir en la conocida frase de DeMarco [DeM82] “No se puede controlar lo que no se puede medir”. Por otra parte, una buena gestión supone predecir el comportamiento futuro de los productos y de los procesos del software. La predicción supone contar con datos apropiados y fiables, por lo que según Fenton [FP97], “No se puede predecir lo que no se puede medir”. Hay cuatro razones para medir procesos software, productos o recursos:

- Medir para caracterizar
- Medir para evaluar
- Medir para predecir
- Medir para mejorar

Medir para caracterizar mejora la comprensión y establece la línea base para futuras comparaciones. Medir para evaluar ayuda a determinar el estado respecto a unos planes iniciales y para comprobar si se han alcanzado los objetivos de calidad. Medir para predecir ayuda a planear, sirve para ganar comprensión de relaciones entre procesos y productos, y construir modelos con esas relaciones, de forma que los valores observados en algunos atributos pueden servir para predecir otros. Medir para mejorar permite identificar ineficiencias, oportunidades, causas etc. interpretando la información cuantitativa de las medidas.

La medición forma parte de las disciplinas científicas y de la ingeniería. Es difícil evaluar las técnicas y métodos de ingeniería del software sin utilizar la medición. En particular, en las tareas de mantenimiento del software son de gran utilidad medidas de diferentes características del producto software, que repercuten en la facilidad de mantenimiento del mismo.

El proceso de medición permite asociar un atributo de una entidad real a un valor, mediante reglas bien definidas [FP97]. En el software, las entidades cuyos atributos pueden interesar medir se clasifican en tres grandes grupos: procesos, productos y recursos. Por otra parte, los atributos a medir se clasifican en: atributos internos, aquellos que se pueden medir en términos de la propia entidad, y externos, aquellos que reflejan la relación de la entidad con el entorno.

En la década de los 90 con el auge del desarrollo orientado a objetos (OO), se empezaron a definir colecciones de métricas a modo de indicadores del uso de mecanismos propios

de este paradigma. Las características que se pueden medir en entidades de orientación a objetos son numerosas, pero muchas de ellas carecen de interés, por ello algunos autores sugieren recomendaciones a seguir cuando se definen métricas [BEM95, Fen94, FP97, FN99, Mor01, BMB02, CPG01] que se pueden sintetizar así:

- Las métricas se definen en función de un objetivo claro.
- Las métricas deben validarse teóricamente para dilucidar si miden realmente los atributos que se pretendían medir, lo que permitirá entre otras cosas, conocer la escala de medida que limitará las operaciones posibles con las mediciones.
- Las métricas se deben validar empíricamente, para conocer su grado de utilidad en la obtención de medidas de atributos de calidad externos que no son directamente medibles, o para gestionarlos.
- La obtención de las mediciones debe ser fácil, y a ser posible, automatizada con herramientas adecuadas.

Las teorías y modelos de calidad del software tienen importancia porque nos ayudan a entender qué factores repercuten en ella, con una aplicación práctica evidente pues proporcionan guías a los desarrolladores para obtener productos mejores y más fiables. En una época en la que el software está presente en todos los ámbitos tanto de la enseñanza, comunicaciones, económicos, salud, políticos, transporte etc., producir software de calidad es crucial para la sociedad [KP96]. El problema fundamental en cuanto a la calidad del software es precisar qué significa calidad y encontrar modelos y medidas adecuadas, lo cual no es trivial [FP97]. Una de las primeras propuestas para modelar la calidad se hizo en la década de los 70 [JAM77, BBL76]. McCall propuso un modelo jerárquico que define la calidad de un producto software a partir de tres elementos: factores, criterios y métricas. Aunque dicho modelo presentara limitaciones por su visión estática del producto software y por la falta de nexos, correlacionando los elementos del modelo con las métricas, ha servido de patrón inicial y su infraestructura modificada se ha utilizado posteriormente en estándares [ISO01, KP96].

El estándar ISO 9126 [ISO01] es uno de los más extendidos, y sirve como punto de partida para otros trabajos que lo amplían y mejoran [BBAA03, Moo05]. La revisión 2001-2002 del estándar ISO 9126 propone tres puntos de vista o tipos de calidad, junto con las medidas correspondientes, con el fin de flexibilizar el modelo de calidad del software: calidad interna, calidad externa y calidad en uso. La técnica propuesta sólo permite tratar un punto de vista cada vez, no de forma concurrente, aunque sobre esta idea se han construido técnicas y marcos para integrar los tres tipos de calidad del software a lo largo del ciclo de vida en una matriz de tres dimensiones [BBAA03].

Este estándar incluye atributos externos de calidad del producto como funcionalidad, fiabilidad, facilidad de uso, eficiencia, facilidad de mantenimiento y portabilidad (ISO 9126, 2001), medibles en las etapas finales del desarrollo. En general, los modelos de calidad se proponen sin que existan suficientes evidencias empíricas que demuestren la relación que hay entre atributos externos, atributos internos y métricas, lo que impide concluir si la definición de calidad es adecuada o no. Otro problema que no es menor, se plantea cuando se quiere medir un atributo externo que depende de una colección

de atributos internos con las correspondientes métricas asociadas.

En conclusión, producir software de calidad es crucial para la sociedad y su multidimensionalidad da lugar a que los problemas planteados se resuelvan en diferentes direcciones, dependiendo del interés particular [KP96]. Unos modelos jerarquizan la calidad ISO 9126 y otros no [Dro96], pero hay un axioma en el que se fundamentan todos:

“Los atributos externos de calidad de un producto están determinados por las propiedades internas y medibles de los mismos.”

1.3. Experimentación en ingeniería del software

La experimentación es un instrumento necesario para investigar. En ingeniería del software el paradigma experimental todavía se encuentra en las primeras etapas de desarrollo en lo referente a diseño de experimentos, extracción del conocimiento útil y extrapolación de ese conocimiento [DF00]. La experimentación no es fácil de llevar a cabo. En [BSL99] los autores comentan que una de las razones de esta dificultad estriba en la gran cantidad de variables del contexto que se deben tener en cuenta, con la complejidad consiguiente que esto puede añadir a los modelos teóricos o al diseño del experimento.

Cuando se plantea un estudio empírico, siempre hay un conjunto de elementos del mundo real del que se necesita conocer cierto comportamiento, que se expresa a través de una hipótesis. Ésta se quiere aceptar o rechazar tomando como base los resultados observables y medibles. En [DF00, JM01, WBM91] se presentan tres tipos de estrategias empíricas que pueden ayudar para realizar esta tarea: experimentos, casos de estudio y encuestas.

Las encuestas normalmente se hacen en retrospectiva cuando, por ejemplo, una técnica o herramienta se ha estado usando durante cierto tiempo. Tanto los datos cualitativos como cuantitativos se recogen a través de cuestionarios o entrevistas. Permiten obtener conclusiones descriptivas y exploratorias.

Los casos de estudio son estudios observacionales, esto es, investigan fenómenos en el contexto real cotidiano conforme se desarrollan. En la ingeniería del software se suelen utilizar para estudiar fenómenos a gran escala, como el desarrollo completo de procesos, monitorización de proyectos, o para evaluar métodos o herramientas. Su propósito está orientado normalmente a analizar un determinado atributo o establecer relaciones entre diferentes atributos.

Los experimentos, a diferencia de los casos de estudio, permiten gran nivel de control. Su objetivo es variar una o más variables y controlar otras, para analizar el efecto que tienen sobre una determinada variable de salida. Se ejecutan off-line (en laboratorios).

Otro aspecto importante para realizar experimentación en ingeniería del software es la elección de un proceso que asista su desarrollo. El proceso experimental propuesto en [WRH⁺00], consta de seis etapas principales:

1. Definición
2. Planificación
 - a) Selección de contexto
 - b) Formulación de la hipótesis
 - c) Selección de sujetos
 - d) Diseño del experimento
 - e) Instrumentación
3. Operación
4. Análisis e interpretación
5. Evaluación de la validez
6. Presentación y empaquetamiento

Bajo este prisma de experimentación existen trabajos diversos. Entre ellos hemos revisado aquellos relacionados con el mantenimiento y la validación de métricas de clase, pues son aspectos relevantes del contexto de esta tesis. Entre los trabajos que validan métricas, se encuentran los siguientes:

- En [BEM95, BWH98, EBGR01, EBG⁺02, HCN98, SL08] estudian la relación entre las métricas y la predisposición de las clases al error. Los resultados apuntan como métricas relevantes las de tamaño, acoplamiento y número de métodos, aunque cuando se trata de predecir la predisposición a error durante el mantenimiento, los modelos de predicción no tienen suficiente precisión [SL08]. Además hay que tener en cuenta que el tamaño puede ser un factor de confusión, esto es, las métricas están correlacionadas con el tamaño y éste puede ser el que realmente produce más errores; así que la correlación entre métricas y predisposición a error sería solo aparente, hecho que se corrobora en [EBGR01] cuando se realiza el estudio de correlación entre métricas y predisposición de las clases a defectos, ajustando por el tamaño.
- En [KSW99] los sujetos realizan una evaluación de modelos sirviéndose de una colección de métricas, y los resultados señalan a las métricas de Chidamber y Kemerer (CK) [CK91] y las de Brito e Abreu y Carapuca (MOOD) [eAC94] como las más útiles para evaluar dichos modelos.
- En [SPD⁺05] se estudia la capacidad de las métricas de CK para caracterizar los proyectos software. Las conclusiones son de utilidad tanto para investigadores como para desarrolladores, pues detectan que existe fuerte colinealidad entre algunas métricas, esto es, las métricas CK contienen información redundante. Además señalan que cuando se utilizan métodos estadísticos para analizar los resultados hay que tener en cuenta que algunas métricas presentan poca variabilidad.

En resumen, de los experimentos examinados cuyo objetivo es validar empíricamente métricas de clases o de modelos OO, una gran parte estudian la correlación con la pre-

disposición de las clases al error, encontrándose resultados significativos para métricas de tamaño (a nivel de código), acoplamiento y número de métodos.

Otros experimentos se refieren al mantenimiento del software a diferentes niveles, desde modelos de diseño hasta código. El mantenimiento del software es relevante teniendo en cuenta los recursos que consume [Pig96]. A continuación se describen algunos experimentos agrupándolos por similitud de objetivos:

- En [ABF04] se estudia la relación entre métricas de acoplamiento dinámico de un sistema con la tasa de cambios durante el mantenimiento del mismo; los resultados observados en un sistema Java son favorables al considerar las métricas como indicadores de la predisposición de las clases al cambio, contribuyendo especialmente las métricas que miden el acoplamiento.
- En [AS04] se investiga el efecto que tiene el control centralizado vs. el delegado, asociados a diseño procedural y orientado a objetos, respectivamente, en el mantenimiento de modelos de diseño. Los resultados son favorables al uso de control delegado cuando los desarrolladores son expertos, pero es peor para los inexpertos. Luego la experiencia de los desarrolladores es un parámetro relevante en el mantenimiento de los modelos.
- En [BBD01] se realiza un experimento para evaluar la validez de guías de diseño basadas en los principios de Coad y Yourdon (1991), para garantizar la calidad del software. En él se estudian dos atributos de facilidad de mantenimiento, la facilidad de comprensión y la facilidad de modificación, cuando se practican “buenas” o “malas” guías de diseño. El seguimiento o no de las guías se evalúa con métricas de CK relativas a la profundidad de herencia, número de hijos y acoplamiento entre objetos. Los resultados muestran que los modelos de diseño realizados siguiendo los principios de Coad y Yourdon son más fáciles de mantener que los realizados sin seguirlos.
- Un estudio similar al anterior se lleva a cabo en [DSA⁺04] para evaluar el efecto de la heurística para detectar el defecto de diseño “God Class”, con la facilidad de entendimiento (evaluada subjetivamente) y con la facilidad de mantenimiento (evaluada subjetiva y objetivamente) de modelos de diseño. Los resultados proporcionan suficiente evidencia del efecto beneficioso de la heurística en ambos factores de calidad, facilitando además un uso adecuado de la herencia.

En resumen, hay diferentes parámetros que influyen en el mantenimiento de modelos o sistemas de software. Las actividades relacionadas con el mantenimiento se facilitan cuando usamos ciertas heurísticas [BBD01, DSA⁺04], o estilos de diseño. Además es interesante disponer de indicadores de la facilidad de mantenimiento [AS04, BVT03, FN01, GOPR01] aunque en casi todos los casos se debe disponer del código y sabemos que, a ese nivel, la facilidad del mantenimiento depende del tipo de lenguaje utilizado en la representación del software [BLYP04].

1.4. Definición del contexto

En esta tesis se pretende utilizar medidas de código para predecir la existencia de ciertos tipos de defectos de diseño en un sistema, continuando con la línea mostrada en trabajos como [Mar02a]. Este autor indica que la predicción de defectos se convierte en un aspecto relevante desde el punto de vista de la calidad del software, ya que los defectos de diseño pueden considerarse como atributos internos del software relacionados con la característica de facilidad de mantenimiento. En el contexto de este trabajo se abordará la predicción de defectos de diseño a partir de mediciones sobre entidades de producto, particularmente entidades de código, implementadas bajo el paradigma de la orientación a objetos (paquetes, clases y métodos). Los defectos de diseño se identificarán a través de mediciones, obtenidas a partir de un conjunto de métricas de orientación a objetos extraídos de la literatura existente, como son:

- Métricas de subsistemas: Robert Martin [Mar94], Brito e Abreu [eAC94].
- Métricas de clases: Chidamber y Kemerer [CK91], Lorenz y Kid [LK94].
- Métricas de complejidad basada en métodos: McCabe [McC76].
- Otras recogidas por Piattini [PG02].

Se empleará experimentación en ingeniería del software para analizar la influencia en la detección de defectos de una nueva dimensión relacionada con la naturaleza de diseño de la entidad medida. Se entiende por naturaleza de diseño de la entidad aquella que se expresa con estereotipos estándar de clasificadores UML. Como objetivo final se pretende formular algoritmos de detección de defectos usando métricas de código y considerando la naturaleza de la entidad. Se propone utilizar casos de estudio realizados siguiendo el proceso experimental propuesto en [WRH⁺00]. La selección del contexto se basará en códigos orientados a objetos reales que serán medidos con herramientas software de medición de código. Los diseños de los experimentos se basará en estudios de correlaciones de datos, técnicas de contraste de hipótesis estadísticas [WBM91, SC88] y técnicas de clasificación binaria para predicción [Qui86, PW90, Gon08].

1.5. Objetivos de investigación

Objetivo de la tesis:

Desarrollar nuevos métodos y técnicas para mejorar la interpretación de las medidas de código cuando se usan en la detección de defectos de diseño en orientación a objetos.

El foco central de esta tesis es evaluar la calidad de sistemas software O.O., desde el punto de vista de defectos de diseño. Para ello, estableceremos un puente entre las definiciones textuales de defectos y definiciones más precisas, basadas en reglas, compuestas por combinaciones de métricas y sus valores umbrales.

Este objetivo general le alcanzaremos a través de dos objetivos:

Objetivo 1 Mejorar las interpretaciones de métricas orientadas a objetos obtenidas automáticamente sobre entidades de código, añadiendo nueva información relacionada con la naturaleza de la entidad medida. Se entiende por naturaleza de la entidad la indicada a través de estereotipos estándar de clasificadores UML.

Este objetivo comprende los siguientes objetivos parciales:

1. Buscar y estudiar el diseño subyacente de herramientas software que calculan métricas de código.
2. Proponer un diseño genérico para un framework de cálculo de métricas.
3. Seleccionar un conjunto de métricas obteniendo valores umbrales, para identificar medidas anómalas sin considerar la naturaleza de la entidad a medir.
4. Definir y aplicar un nuevo proceso de medición utilizando las métricas elegidas, considerando la naturaleza de la entidad para obtener valores umbrales que identifiquen medidas anómalas.
5. Comprobar de manera empírica, mediante experimentación, que usar la naturaleza de diseño de la entidad para obtener los valores umbrales, mejora la precisión en la identificación de medidas anómalas.
6. Definir un prototipo de herramienta software, que calcule métricas de código, incorporando la naturaleza de la entidad en el proceso de medición.

Objetivo 2 Mejorar las predicciones para detectar defectos de diseño mediante el uso de herramientas que incorporan cálculo de métricas de código, y consideran la naturaleza de la entidad.

Este objetivo comprende los siguientes objetivos parciales:

1. Buscar y estudiar el diseño subyacente de herramientas software de detección de defectos.
2. Definir un proceso de detección de defectos, usando valores umbrales dependientes de la naturaleza de la entidad, y considerando la subjetividad del auditor.
3. Seleccionar un conjunto de defectos y proporcionar clasificadores binarios (basados en reglas o árboles de decisión) a partir de métricas de código y la naturaleza de la entidad medida.
4. Comparar de manera empírica, mediante experimentación, nuestro clasificador binario de detección de defectos frente a otras herramientas de detección.
5. Definir un prototipo de herramienta software de detección de defectos, que incorpore las nuevas tareas del proceso de detección de defectos.

1.6. Estructura de la tesis

El resto del contenido de esta tesis se ha estructurado en cuatro capítulos. El primero (Cap. 2) aborda una revisión del estado del arte sobre la gestión de defectos de diseño. En el segundo (Cap. 3) se presenta nuestra propuesta de proceso de medición modificado y su validación empírica. En el siguiente capítulo (Cap. 4) proponemos un proceso de gestión de defectos y su validación empírica. En el último capítulo (Cap 5), anterior a las conclusiones, se presentan los diseños basados en frameworks y los prototipos de herramientas software, que sirven para automatizar nuestros procesos, de medición y de gestión de defectos.

En la Tabla 1.1 se muestra una relación entre los objetivos y los capítulos en los cuales se van a tratar.

Tabla 1.1. Tabla cruzada de capítulos y objetivos

Cap./ Obj.	Cap. 2	Cap. 3	Cap. 4	Cap. 5
Obj. 1.1	X			
Obj. 1.2				X
Obj. 1.3		X		
Obj. 1.4		X		
Obj. 1.5		X		
Obj. 1.6				X
Obj. 2.1	X			
Obj. 2.2			X	
Obj. 2.3			X	
Obj. 2.4			X	
Obj. 2.5				X

1.6.1. Capítulo 2: Revisión de defectos de diseño y su gestión

En este capítulo hacemos la revisión que nos permitirá establecer el estado del arte con respecto a la detección de defectos de código, cuando se usan métricas. Dicho estudio nos ha permitido determinar y refinar los objetivos desde el inicio de este trabajo.

Los objetivos abordados en este capítulo son el 1.1 y 2.1, y se resumen en, buscar y estudiar el diseño subyacente de herramientas software de medición y de detección de defectos.

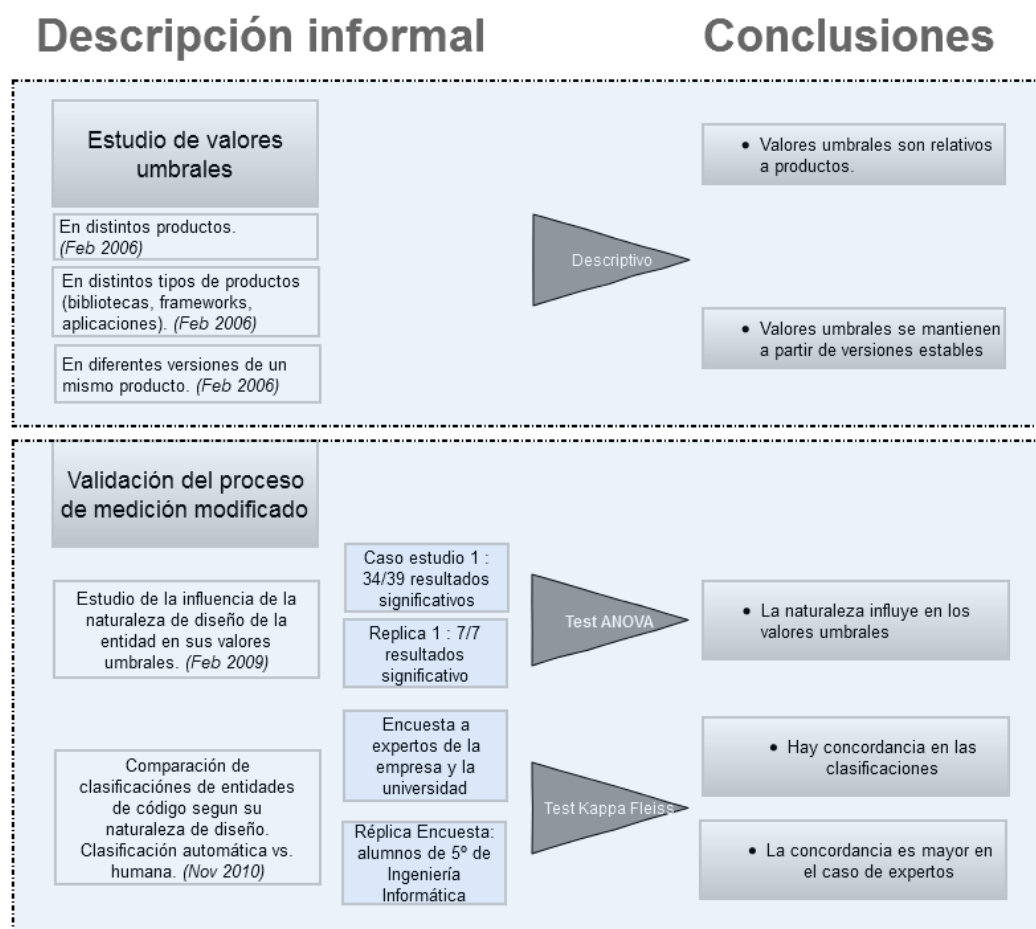
1.6.2. Capítulo 3: Proceso de medición modificado y su validación

En el proceso de medición de software se identifican mediciones anómalas usando valores umbrales de diferentes métricas. En el caso particular de la medición de código, se utilizan los mismos valores umbrales independientemente de la naturaleza del problema que resuelve la entidad a medir. En este capítulo se abordan las modificaciones en el

proceso de medición para incorporar esta nueva variable. Por tanto el proceso debe incluir actividades relacionadas con la clasificación de entidades según su naturaleza de diseño.

Nuestra hipótesis de partida es que los valores umbrales de las medidas están condicionados por la naturaleza del problema que resuelve dicha entidad de código, entendiendo por naturaleza aquella que se expresa con estereotipos estándar de clasificadores UML. Para corroborar esta hipótesis, se realizan estudios observacionales, esto es, investigar este fenómeno en el contexto de proyectos reales de desarrollo del software. En la Fig. 1.1 se muestra un resumen de la experimentación realizada en este capítulo, que incluye una breve descripción de los casos de estudio y las conclusiones obtenidas.

Figura 1.1. Resumen de la experimentación realizada en el Capítulo 3



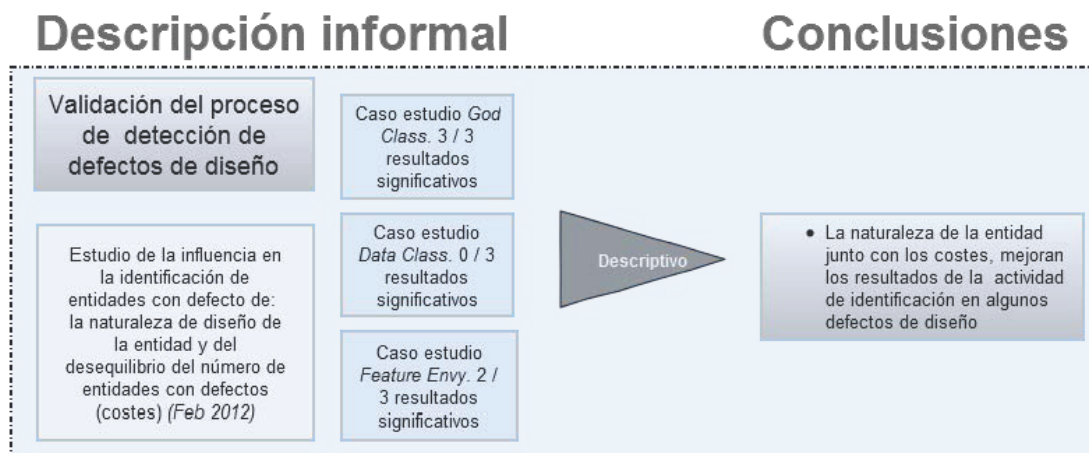
1.6.3. Capítulo 4: Proceso dinámico de gestión de defectos

La detección de defectos de diseño tiene dos características intrínsecas, por un lado, la subjetividad del auditor, y por otro, las fuentes de información utilizadas para su detección. Dependiendo del tipo de defecto: tamaño, duplicación, lógica condicional, etc se pueden necesitar diferentes fuentes de información para su detección. Un proceso que asista la detección de defectos debe considerar ambas características.

En este capítulo, además de la definición de un proceso de detección de defectos, se presenta su aplicación en tres defectos de diseño concretos: *God Class*, *Feature Envy* y *Data Class*. En la aplicación se incluye la nueva información de la naturaleza de diseño de la entidad de código y medidas de eficiencia (precisión vs. recuperación) para evaluar las reglas de detección aplicadas.

Además se realiza un estudio empírico para comprobar la influencia de esta nueva variable denominada naturaleza de diseño de la entidad, en la actividad de identificación de defectos de diseño. En la Fig. 1.2 se presenta un resumen de la experimentación realizada en este capítulo, que incluye una breve descripción de los casos de estudio y las conclusiones obtenidas con cada uno.

Figura 1.2. Resumen de la experimentación realizada en el Capítulo 4



1.6.4. Capítulo 5: Framework para la medición y gestión de defectos

La mayoría de las herramientas que existen para calcular métricas de código y detectar defectos de diseño, dependen del lenguaje utilizado. Pese a que es una solución válida, se debe recordar que la mayoría de las métricas y defectos de diseño, y en mayor medida los orientados a objetos, toman como punto de partida que son independientes del lenguaje. Por ello se propone un *framework* [FSJ99] para utilizarse y extenderse tanto

con métricas y defectos ya definidos, como con nuevas métricas y nuevos defectos. De esta forma se permite el trabajo con la independencia del lenguaje de programación.

Los procesos deben estar asistidos por herramientas que automaticen sus tareas. Como prueba de concepto, en este capítulo, se presentan tres prototipos de herramientas [Lóp12b], uno relacionado con la modificación del proceso de medición especificado en el Cap. 3. Los otros dos, relacionados con el proceso de gestión de defectos especificado en el Cap. 4. El objetivo del prototipado es doble, por un lado, poder observar la interacción del usuario con las actividades de los procesos propuestos. Por otro, validar nuestros diseños basados en *frameworks*.

REVISIÓN DE DEFECTOS DE DISEÑO Y SU GESTIÓN

Este capítulo presenta un recorrido histórico sobre la investigación en gestión de problemas de diseño, en software orientado a objetos. A partir de la revisión bibliográfica relativa a defectos de diseño, se ha elaborado un resumen histórico de dicha investigación. Con el fin de facilitar la comprensión, en este capítulo proponemos una terminología común para los problemas que se estudian, que denominaremos *defecto de diseño*.

Además de revisar las publicaciones sobre la gestión de defectos de diseño, se presenta un estudio detallado de las herramientas de defectos de diseño 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 conocer los aspectos más importantes de la gestión de defectos de diseño. Los desarrolladores de herramientas pueden utilizarlo para comparar y mejorar sus herramientas, y los desarrolladores de software pueden utilizarlo para evaluar qué herramienta o técnica es la más apropiada para sus necesidades.

La taxonomía propuesta en este estudio se ha elaborado en colaboración con Javier Pérez, Naouel Moha y Tom Mens. Una versión previa del estudio y de la taxonomía presentada en esta tesis se encuentra disponible como informe técnico [PLMM11]. Además en [MSPC11] se aplica dicha taxonomía sobre un conjunto de herramientas de gestión de defectos.

En las últimas secciones del capítulo se aplica la taxonomía para caracterizar un par de herramientas de detección de defectos, InCode y JDeodorant, que serán utilizadas en el Cap. 4 para obtener conjuntos iniciales de entidades con defectos de código. Por último, se aplicará la taxonomía para caracterizar el contexto de trabajo de esta tesis.

2.1. Catálogos de defectos de diseño

En esta sección se presenta una revisión de los trabajos relacionados con la definición, catalogación y detección de defectos de diseño. A partir de ella se obtiene tanto la motivación como el ámbito de aplicación de este trabajo.

Webster [Web95] escribe el primer libro sobre defectos en el contexto de programación orientada a objetos, catalogándolos como conceptuales, de codificación, políticos y de garantía de calidad. Riel [Rie96] define 61 heurísticas de diseño que servían para caracterizar un buen programa software, posibilitando a los desarrolladores de software evaluar la calidad de sus sistemas manualmente, y proporcionar las bases para el diseño e implementación de calidad de estos sistemas.

Fowler y Beck recopilan 22 defectos de código en [FBB⁺99], que sugieren dónde y cuándo los desarrolladores de software deberían aplicar las operaciones de mantenimiento conocidas como refactorizaciones. Estos defectos, conocidos como “bad smell”, son descritos con un estilo informal, que es apto para procesos de inspección cuya finalidad es detectar defectos en el código fuente. Wake [Wak03] clasifica los defectos descritos en este catálogo en “defectos dentro de clases” y “defectos entre clases”. La primera categoría a su vez se divide en defectos medibles, convención de nombres, complejidad innecesaria, duplicación y lógica condicional. La segunda se divide en datos, herencia, responsabilidad, adecuación al cambio y bibliotecas de clases. Además, mejora la definición textual de los defectos aplicando una plantilla de definición con los campos: síntomas, causas, qué hacer, ventajas, comentarios y contradicciones.

Brown et al. [BMMM98] proporcionan otra fuente de especificaciones de defectos, conocidos como “antipatterns”. Los autores se centran en todo el proceso de desarrollo de sistemas OO y describen textualmente 41 “antipatterns”, 14 de éstos están relacionados con el desarrollo del software, 13 con la arquitectura y los otros 14 con la gestión de proyectos. En la categoría de desarrollo del software se encuentran los defectos bien conocidos “Blob” y “Spaghetti Code”. Las categorías de arquitectura y gestión de proyectos se salen del contexto de este trabajo.

Kerievsky [Ker04] añade algunos nuevos defectos al catálogo de Fowler y Beck, como: “Solution Sprawl”, “Oddball Solution”, “Indecent Exposure” y “Combinatorial Explosion”. Además, relaciona los defectos con refactorizaciones y patrones de diseño [GHJV95]. Algunos defectos pueden aparecer por el uso incorrecto de patrones de diseño, pero por otro lado otros se pueden corregir aplicando algún patrón de diseño.

Todas estas referencias y catálogos proporcionan una amplia visión sobre heurísticas, defectos y antipatrones dirigidas a una audiencia con propósitos educacionales. Sin embargo, la inspección manual de código para buscar defectos en base sólo a descripciones textuales es una actividad que consume mucho tiempo y está expuesta a errores en sus resultados. Algunos trabajos han analizado y recopilado los defectos de diseño con más detalle, estructurando sus descripciones para facilitar la detección humana o automática.

Roock et al. [SR06] describen defectos arquitectónicos basados en principios de diseño,

clasificando estos defectos en categorías respecto de: la jerarquía de herencia, el grafo de dependencias, los paquetes, los subsistemas y las capas.

Algunos de estos defectos tienen una alta correlación con ciertas métricas. Ejemplo de esto es el caso del defecto “dependencias entre capas” para el que se puede utilizar la métrica de “ciclos entre paquetes” configurada previamente para trabajar únicamente con capas, o la métrica “distancia a la secuencia principal” [Mar94] para medir la calidad del diseño del grafo de dependencias.

Lanza y Marinescu [LM06] presentan un catálogo de defectos llamados “disharmonies”, junto con la definición de las estrategias para su detección y recomendaciones para su corrección. Su enfoque introduce reglas basadas en métricas para detectar las desviaciones de los buenos principios de diseño. El término “disharmony” se puede entender más como un intento para unificar la terminología que como una clase de defecto en sí mismo, 7 de los 11 defectos presentados en [LM06] se corresponden con problemas citados ya en [FBB⁺99]. La definición de las estrategias de detección son un avance hacia la especificación precisa de los defectos de diseño que permite la automatización de su detección, como se manifiesta en las herramientas InCode e InFusion [Int08].

Trifu [Tri08] describe un conjunto de 10 defectos de diseño, bajo el nombre de “design flaws”, y proporciona “restructuring patterns” para detectarlos y corregirlos. Los patrones de reestructuración añaden una nueva dimensión ya que relacionan los defectos con la intención del diseño de la entidad. De esta forma, se tiene en consideración los casos excepcionales de identificación de defectos que dependen de la intención de diseño que subyace en la entidad.

Moha et al. [MGDL10] proponen un método y una técnica para especificar, detectar y visualizar defectos de diseño. Utilizan un lenguaje específico de dominio para permitir al usuario especificar los defectos. Las especificaciones se basan en métricas y consultas sobre la estructura del código. Estas especificaciones se transforman en código Java auto-generado que puede ser compilado y ejecutado para buscar defectos. El proceso de especificación es manual, pero la detección y visualización están completamente automatizados a través de DECOR [Tea09], la herramienta que implementa esta propuesta.

La Tabla 2.1 muestra un resumen de esta revisión sobre defectos de diseño. Las columnas representan diferentes aspectos utilizados para caracterizar los trabajos descritos anteriormente, y las filas representa la caracterización de cada uno. En [PLMM11] está descrita una inspección más detallada respecto al estado del arte en la gestión de defectos.

2.2. Visión general del estudio

En esta sección se presenta un estudio detallado de las distintas soluciones en la gestión de defectos de diseño. Además se proporciona un marco de referencia para comparar y analizar las técnicas y herramientas de gestión de defectos, tanto actuales como futuras.

El estudio se puede utilizar para una gran variedad de propósitos. Entre otros, puede ayudar a los desarrolladores a elegir una solución particular que se adecúe a sus nece-

Tabla 2.1. Resumen de los trabajos de defectos

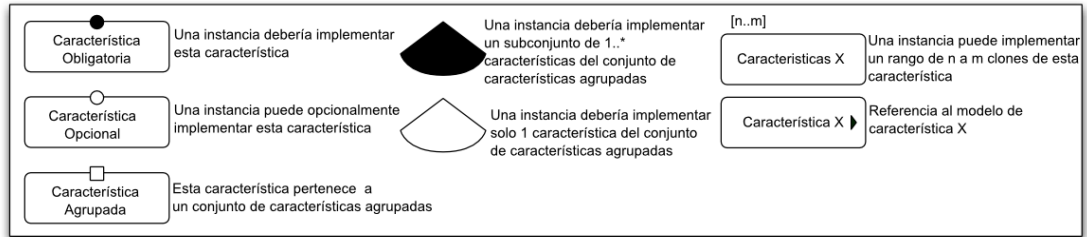
Autores	Nomenclatura	Formato de descripción del defecto	Herramienta asociada	Clasifica los defectos
Webster [Web95]	Pitfalls	Textual	No	Si
Riel [Rie96]	Design Heuristics	Textual	No	Si
Fowler y Beck [FBB ⁺ 99]	Bad Smells	Textual	No	No
Wake [Wak03]	Bad Smell	Plantillas	No	Si
Brown et al. [BMMM98]	Antipatterns	Plantilla	No	Si
Kerievsky [Ker04]	Code Smell	Textual	No	No
Roock et al. [SR06]	Architecture Smells	Textual	JDepend [Con99]	Si
Lanza y Marinescu [LM06]	Disharmony	Expresiones lógicas basadas en métricas. Estrategias de detección.	InCode e InFusion [Int08]	Si
Trifu [Tri08]	Design flaws	Restructuring patterns	CodeClinic	No
Moha et al. [MGDL10]	Antipattern, Bad Smell	Lenguaje de dominio basado en métricas y consultas estructurales	DECOR [Tea09]	No

sidades, puede ayudar a los desarrolladores de herramientas a valorar las fortalezas y debilidades de su herramienta comparada con otras, y puede ayudar a los investigadores a identificar las tecnologías o herramientas que pueden ser mejoradas mediante la aplicación de nuevas técnicas y formalismos.

2.2.1. Notación del modelo de características

Como ayuda para guiar el estudio detallado, se utiliza la notación visual denominada *diagrama de características*, que está inspirado en la utilización que hacen Czarnecki y Helsen para presentar un estudio detallado sobre modelos de transformación [CH06]. Los *diagramas de características* son una representación visual de modelos de características. En la literatura existen diferentes símbolos y notaciones para este tipo de diagramas, la notación utilizada en este estudio se muestra en Figura 2.1.

El modelado de características [CE00] es la actividad que permite modelar las propieda-

Figura 2.1. Notación del diagrama de características utilizado en el estudio.

des comunes y variables de conceptos y sus interdependencias, organizándolos en un modelo de referencia denominado modelo de características. Este modelo se utiliza para representar una jerarquía de características, mostrando las propiedades comunes y variables de las instancias de los conceptos y las dependencias entre las características variables.

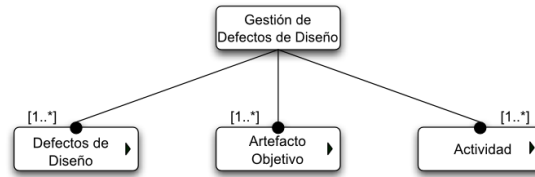
Los modelos de características son una manera intuitiva para representar familias de sistemas, o un concepto como la gestión de defectos de diseño, a través del análisis de las similitudes y diferencias entre la gran variedad de aproximaciones que lo abordan.

Las características son propiedades importantes de un concepto, y la característica principal del modelo es la que describe una familia o concepto, dentro de un dominio dado, obtenida a través del análisis y especificación de ocurrencias particulares. Las características también sirven para capturar y modelar el conocimiento y terminología de ese dominio. Los fundamentos para el modelado de características se pueden encontrar en el libro de Czarnecki y Eisenecker [CE00].

Durante el análisis de las distintas soluciones en la gestión de defectos de diseño, hemos detectado muchos puntos en común compartidos entre todas ellas, o entre algunos subgrupos. Por ejemplo, todos los enfoques estaban dirigidos a un cierto tipo de artefacto objetivo. De la misma manera, hemos identificado diferencias relevantes que pueden ser utilizadas para caracterizar cada aproximación. A modo de ejemplo (véase la Sec. 2.4), una solución puede buscar defectos de diseño en el código fuente o en código ejecutable, modelos, etc. Además, dado que todas las herramientas de gestión de defectos de diseño pertenecen a una familia de soluciones, se pueden describir de manera natural con un modelo de características.

Por tanto este estudio detallado y su representación mediante el diagrama de características refleja el estado actual del arte. Las características identificadas en algunas categorías, como “Propiedades del Defecto” en la Sec. 2.3 o “Tipo de Representación” en la Sec. 2.4, se pueden extender en soluciones futuras. Por este motivo, a veces se han añadido a los diagramas de características, elementos que actualmente no se emparejan con ninguna de las soluciones actuales, sino más bien ilustran lo que se considera factible o deseable en un futuro próximo. Por ejemplo, en el caso de “Artefacto objetivo” la multiplicidad que figura es 1..* (Fig. 2.2) cuando los trabajos actuales se enfocan en un solo artefacto.

Figura 2.2. Nivel superior del modelo de características para comparar soluciones en la gestión de defectos de diseño



2.2.2. Características del nivel superior de la gestión de defectos de diseño

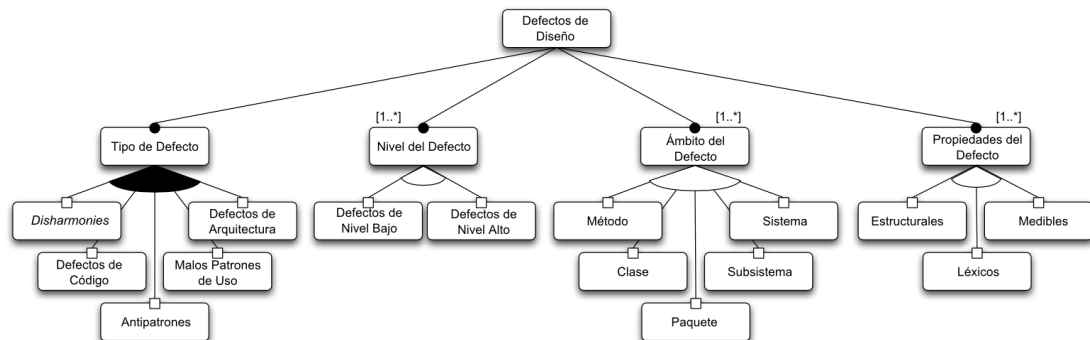
El diagrama de características propuesto permite agrupar, basándose en los puntos comunes, las actividades, herramientas, técnicas o formalismos de la gestión de defectos. La clasificación multidimensional, permite describir y comparar diferentes soluciones basadas en los criterios de interés. En la clasificación no se han considerado propiedades generales como la interoperabilidad, la facilidad de uso o la facilidad de extensión porque pueden ser aplicadas a cualquier clase de herramientas de un dominio de interés. Estas propiedades generales no son específicas o intrínsecas a las distintas soluciones de gestión de defectos de diseño.

Para presentar el estudio de manera estructurada, cada una de las secciones siguientes contiene una discusión sobre las características principales referentes a la gestión de defectos. Se empieza describiendo las características de nivel superior, que constituyen las principales propiedades comunes del campo de estudio. En secciones posteriores se descenderá por el modelo para describir cada una de estas características. La característica raíz del modelo es la gestión de los defectos de diseño, representa cualquier solución o propuesta que trate con defectos de diseño. Una instancia del modelo representa una solución existente, o una no existente que podría ser interesante desarrollar. En la Fig. 2.2 se muestran las tres características de nivel superior para la gestión de defectos de diseño. Describen propiedades comunes y obligatorias para todas las aproximaciones.

Defecto de diseño. Las diferentes soluciones o aproximaciones gestionan una gran variedad de defectos de diseño. La naturaleza de los defectos de diseño de cada aproximación es uno de los puntos de mayor variación.

Artefacto objetivo. Cualquier herramienta de gestión de defectos necesita al menos un artefacto software sobre el cuál se deben observar los defectos de diseño. La característica Artefacto objetivo se utilizará para modelar este (conjunto de) artefacto(s) software.

Actividad. El tercer punto de variación en la gestión de defectos es el conjunto de actividades explícitas soportadas por la aproximación. Un ejemplo de actividad es la especificación del defecto. Cada solución necesita una definición del defecto para poder detectarlo, pero sólo algunas presentan un soporte explícito para la actividad de especificación.

Figura 2.3. Característica defecto de diseño y sus subcaracterísticas

Las siguientes secciones discutirán cada una de estas características en detalle, descomponiéndolas en subcaracterísticas para inspeccionar el estado del arte de la gestión de defectos de diseño.

2.3. Defectos de diseño

Como se introdujo en el diagrama de características raíz en la Fig. 2.2, una solución de gestión de defectos de diseño puede cubrir varios defectos. La característica Defectos de Diseño, representada en la Fig. 2.3, ayuda a describir con más detalle la naturaleza de estos defectos. Esta característica se divide en cuatro subramas o subcaracterísticas. Muchas soluciones están especializadas en:

1. Un cierto tipo de defectos
2. Defectos de diferentes niveles de abstracción
3. Defectos que afectan a varias entidades de programa en un nivel particular de granularidad
4. Defectos que tienen propiedades internas de diferente naturaleza

Esta clasificación de defectos de diseño extiende las propuestas en [MLC06b, MGD10]. Aunque el uso de esta clasificación es establecer un marco de referencia para las propuestas de gestión de defectos de diseño, es útil para clasificar defectos de diseño en sí mismos.

2.3.1. Tipo de defecto

Esta característica describe el tipo de defecto de diseño que se utiliza en una solución. “Tipo de defecto” se refiere al catálogo en el cuál está definido. Algunas propuestas definen sus propios conjuntos de defectos [LM06], pero la mayoría se centran en defectos

descritos en un pequeño número de catálogos. En el diagrama de características de la Fig. 2.3 se representan los catálogos más frecuentemente referenciados.

Marinescu et al. [LM06] definen *disharmonies* como defectos de diseño que afectan a entidades como clases y métodos. La particularidad de estas *disharmonies* es que los efectos negativos sobre la calidad de los elementos de diseño se pueden percibir considerándolos de manera aislada. Establecen tres aspectos que contribuyen a identificar la *disharmony* en una entidad: su tamaño, su interfaz y su implementación. iPlasma [Gro], InCode e InFusion [Int08] son tres herramientas que detectan estas *disharmonies* usando métricas orientadas a objetos con valores umbrales personalizados.

Muchas de las aproximaciones estudiadas se centran en los defectos de código especificados en el catálogo de Fowler [FBB⁺99], algunas son [AS09, CLMM06a, TCC08, Mun05, Sem07]. Munro [Mun05] propone heurísticas basadas en métricas con valores umbrales para detectar defectos de código, siendo similar a las estrategias de detección de Marinescu [Mar02a]. Alikacem y Sahraoui [AS09] proponen un lenguaje para detectar defectos y violaciones de principios de calidad en sistemas orientados a objetos. Este lenguaje permite la especificación de reglas usando métricas, herencia, o relaciones de asociación entre clases, de acuerdo con las expectativas del usuario. También utiliza lógica difusa para expresar los valores umbrales de las condiciones de las reglas.

El método DECOR desarrollado por Moha et al. [MGDL10] se centra principalmente en los antipatronos de Brown [BMMM98]. Otro ejemplo de aproximación que identifica antipatronos y defectos de código en sistemas Java es la herramienta Analyst4j [Ana08]. La herramienta usa métricas para la identificación.

Algunas aproximaciones usan *patrones de diseño* [GHJV95] para buscar defectos de diseño, oportunidades para aplicar patrones de diseño o detectar malas aplicaciones de patrones [GAA01, Ker04, TM03]. Guéhéneuc et al. [GAA01] utilizan los patrones de diseño como estructuras de referencia, y detectan los defectos de diseño como intentos fallidos de aplicar patrones de diseño. Para conseguirlo, buscan estructuras que se asemejen a los patrones de diseño pero se desvían ligeramente de ellas. Para corregir estos defectos, sugieren transformar estas estructuras para que su intención coincida con la estructura del patrón de diseño. En [Ker04], Kerievsky propone una solución más manual para detectar y corregir defectos relacionados con patrones de diseño. Para ello, instruye al desarrollador en la búsqueda de estructuras que revelan la mala aplicación del patrón de diseño. Además describe situaciones donde el patrón de diseño está ausente y se podría aplicar, y situaciones donde el patrón del diseño está estorbando y por lo tanto debería eliminarse.

Roock and Lippert [SR06] presentan un catálogo de “defectos arquitectónicos” relacionados con la organización de subsistemas. Entre estos defectos se encuentran problemas relacionados con las dependencias como las dependencias cíclicas. Algunos de estos defectos pueden detectarse con herramientas como [Con99, STA, Tes08].

2.3.2. Nivel del defecto

Otra forma de clasificar los defectos de diseño es mediante la distinción entre defectos de bajo nivel y de alto nivel. Los *defectos de bajo nivel* se centran en un único problema muy específico. Los defectos de código [FBB⁺99] como “*Long method*”, “*Data class*” y “*Large class*” son ejemplos de este tipo de defectos que se refieren a situaciones específicas observadas en el código.

Los *defectos de alto nivel* son defectos de diseño que están compuestos de otros defectos, como los antipatrones. Se centran en una variedad de problemas. Un ejemplo es el antipatrón “*Blob*” [BMMM98], también conocido como “*God class*”, que revela un diseño procedural implementado con un lenguaje de programación orientado a objetos. Esta situación se manifiesta con una gran clase controladora que monopoliza la ejecución del programa y que está rodeada por un número de pequeñas clases de datos con muchos atributos y pocos métodos. Este defecto de alto nivel está compuesto de otros de bajo nivel como “*Data class*” y “*Large class*”.

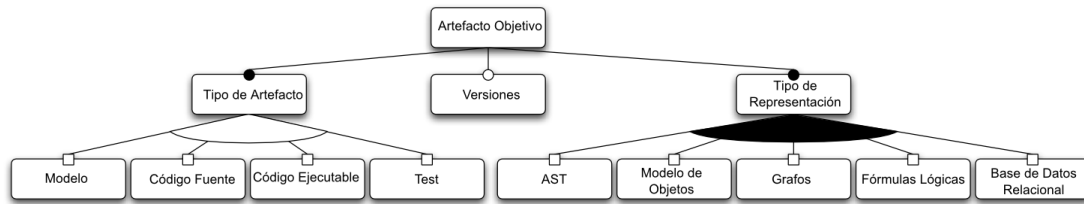
2.3.3. Ámbito del defecto

Esta característica se utiliza para describir la extensión o el ámbito de los defectos soportados sobre los diferentes tipos de entidades involucradas. Para la mayoría de los lenguajes orientados a objetos, los ámbitos son: subsistema, paquete, clase y método. Los defectos de alto nivel normalmente representan problemas de diseño de un gran ámbito, afectando a varias y/o grandes entidades. Por el contrario, los de bajo nivel tienen un efecto sobre un ámbito bien definido y limitado, por ejemplo una única y pequeña entidad. Como se muestra en la multiplicidad asociada a esta característica en la Fig. 2.3, un defecto de diseño puede afectar a varias entidades de diferentes niveles. En la siguiente enumeración se muestra un ejemplo de defecto para cada una de las categorías respecto a la característica ámbito:

- Ámbito de método: *Feature envy*
- Ámbito de clase: *God class*
- Ámbito de paquetes: ciclos entre paquetes
- Ámbito de subsistema: ciclos entre subsistemas

2.3.4. Propiedades del defecto

La naturaleza de un defecto se puede resumir y representar por los indicadores o propiedades utilizadas en su especificación. Estas propiedades se pueden descomponer en: descripciones estructurales del diseño subyacente en el código, especificaciones medibles basadas en métricas y definiciones léxicas basadas en el nombre de las entidades software. Las diferentes soluciones de gestión de defectos de diseño se benefician de estas propiedades con el fin de hacer frente a una actividad en particular. Por ejemplo, en la descripción del defecto *God Class*, la clase grande corresponde a una propiedad

Figura 2.4. Característica artefacto objetivo y sus características

medible que se puede calcular fácilmente contando el número de métodos y atributos, mientras que las clases de datos es una propiedad estructural que consiste en identificar métodos de acceso. Una propiedad léxica en el defecto *God Class* se corresponde con el uso de nombres procedimentales (main, make, create, exec) en las clases afectadas por el defecto. iPlasma [Gro, LM06] utiliza las propiedades medibles de *God Class*, para detectarlo, mientras DECOR [MGDL10, Tea09] añade propiedades léxicas para realizar la detección.

2.4. Artefacto

La característica de artefacto objetivo es el mayor punto de variación que distingue los diferentes enfoques en la gestión de los defectos de diseño. Esta característica se refiere a los artefactos software en los cuales se observan los defectos. En la Fig. 2.4 se muestra esta característica junto con sus subcaracterísticas.

2.4.1. Tipo de artefacto

Cualquier solución de la gestión de defectos, debe centrarse en al menos un tipo de artefacto software. Los tipos de artefactos soportados por una propuesta, y la forma en que se representan internamente, están fuertemente acoplados a los defectos que se pueden gestionar y a cómo pueden ser gestionados.

A menudo las herramientas o técnicas están dirigidas a tratar un único tipo de artefacto. De hecho, no hemos podido encontrar herramientas que soporten muchos tipos de artefactos. A pesar de ello, es factible y deseable construir herramientas que utilicen distintos tipos de artefactos como fuentes complementarias de información, para mejorar los resultados. El proceso de detección manual descrito por Travassos et al. [TSFB99] ilustra que este planteamiento es factible. Para identificar defectos de código, se instruye a los desarrolladores para examinar diferentes tipos de modelos, como descripciones de requisitos, casos de uso, diagramas de clases, descripciones de clases, diagramas de estado y diagramas de interacción.

La descomposición de la característica tipo de artefacto se muestra en la Fig. 2.4 como un conjunto de subcaracterísticas agrupadas. Los tipos de artefactos más comunes

son los códigos fuente y los códigos *bytecodes* o ejecutables. Además, varias soluciones soportan la detección de defectos de diseño analizando modelos software. Una tendencia de las herramientas de modelado es proveer al usuario avisos del modelo, llamados críticas. Las herramientas ArgoUML [Arg] y Together [Bor] soportan esta tendencia.

La gran mayoría de herramientas disponibles están dirigidas a gestionar defectos de diseño en el código. Además, el concepto de defecto de diseño es independiente del lenguaje de programación. Por ejemplo, CheckStyle [Che04] y Hammurapi [Ham07], cargan código Java y buscan violaciones de estándares de código. Reek [Rut] y Roo-di [Roo] buscan defectos en código fuente escrito con el lenguaje Ruby. Algunos autores, como Ciupke [Ciu99] y Sahraoui et al. [SGM00] analizan defectos en C++. FX-Cop [FXC06] y StyleCop [Sty] encuentran desviaciones de las convenciones de código en C#. iPlasma [Gro, LM06], DECOR [MGDL10], InCode e InFusion [Int08] proporcionan un soporte multilenguaje para analizar programas escritos en C++, Java y C#.

Otro tipo de artefacto ampliamente soportado es el código ejecutable (código binario o *bytecodes*). Herramientas como RevJava [Rev] y Stan4J [STA], analizan Java *bytecodes*. La ventaja de estas herramientas es que pueden realizar la detección de defectos sin disponer del código fuente.

Una tendencia emergente en muchas áreas del desarrollo del software, y especialmente en el desarrollo ágil de software, es tratar a los test, más precisamente los test de *script*, como entidades de primera clase. Un número creciente de autores han tratado el problema de los defectos en los test de *script* [Mes07, vMvK01] y realizan propuestas que proporcionan una gestión de estos defectos [NB07, VDDR07].

2.4.2. Versiones

La gestión de defectos se puede beneficiar con la información adicional que se puede extraer del repositorio de control de versiones que almacena múltiples versiones del artefacto objetivo de estudio. Algunos defectos de diseño citados en [FBB⁺99], como *Shotgun Surgery*¹, son identificados más fácilmente analizando la historia del sistema. El soporte para múltiples versiones de un artefacto es una subcaracterística opcional que es aplicable a cualquier tipo de artefacto.

Varias soluciones [GDMR04, RSG08, XS04] proponen usar diferentes versiones del artefacto de entrada. Sus herramientas de soporte a menudo incluyen funcionalidad para acceder a repositorios de control de versiones (CVS, SVN, GIT, etc.), extraen y analizan los artefactos software y sus metadatos desde estos repositorios. Algunas de estas propuestas [GFGP06, LWN07a] incluso afirman que es la única forma de detectar algunos defectos particulares o para obtener una visión más amplia de un determinado problema.

¹Cualquier cambio realizado en una parte del sistema, implica muchos cambios pequeños en otras partes del sistema.

2.4.3. Tipo de representación

Cada propuesta de gestión de defectos utiliza una representación interna del artefacto software. Esta representación se utiliza para analizar y procesar el artefacto objetivo. Esta característica es importante porque existe una dependencia fuerte entre la representación interna y otros aspectos como la técnica, resultados esperados o el soporte de automatización.

Una manera común de representar un artefacto software es por medio de un árbol de sintaxis abstracta (AST). Este tipo de representación es especialmente frecuente en aproximaciones que tratan con código fuente. Una representación típica de AST contiene la información completa disponible en el artefacto. Otras soluciones [TCC08, Sli05, TK04, TCC08] simplifican el AST para mantener sólo la información relevante de la tarea que estén tratando, o incluso aumentan el AST con detalles adicionales para facilitar las actividades de gestión de defectos.

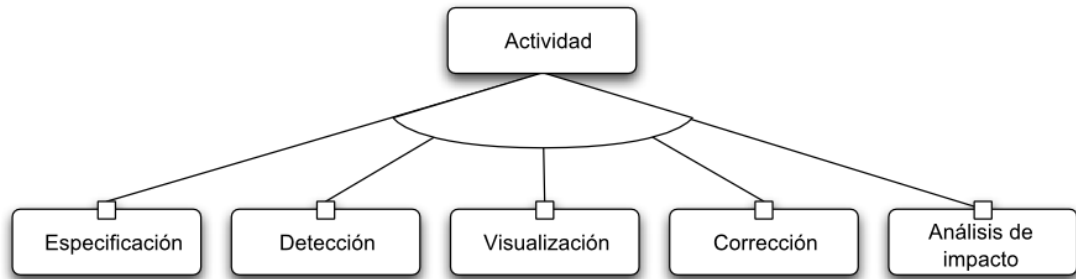
Otras herramientas utilizan una representación basada en algún modelo de objetos. Usan un metamodelo específico del artefacto objetivo, este es el caso de MOON [Cre00, CLMM06a], FAMIX [Tic01, LM06] MEMORIA [Gro]. Generalmente esta representación se usa para simplificar la información del artefacto objetivo, almacenando únicamente la necesaria. El análisis y manipulación del artefacto objetivo se realizan mediante procedimientos programados en una amplia variedad de lenguajes de programación, incluso la personalización de lenguajes de dominio específicos (DSL) [Gué03, MGL⁺09].

También se emplean soluciones basadas en grafos para analizar el artefacto objetivo y buscar defectos [vM02]. La teoría de grafos ayuda en el análisis de artefactos y búsqueda de defectos, y las técnicas de transformación de grafos ayudan a corregir [MVDJ05]. De Lucia et al. [DLOV08] representan las clases con grafos y usan propiedades medibles de estos grafos para encontrar oportunidades de refactorización que mejorarán la cohesión de esas clases.

Las fórmulas lógicas ofrecen un soporte similar para representar los artefactos. Este tipo de representación posibilita el uso de técnicas basadas en lógica para gestionar los defectos de diseño [Ciu99, TM03].

Algunas herramientas almacenan la información extraída del artefacto objetivo en una base de datos relacional. Esto normalmente aumenta la velocidad en las tareas que consultan el modelo software, ya que tienen la ventaja de tener un motor de consultas dedicado y especializado. Este tipo de representación se utiliza internamente por la herramienta citada en [GFGP06, SSL01, TSG04], o incluso se ofrece a los usuarios finales, posibilitándoles analizar artefactos mediante consultas estructuradas. Por ejemplo, con SemmlCode [Sem07], los desarrolladores de software pueden ejecutar consultas sobre el código fuente, utilizando un lenguaje de consulta declarativo llamado .QL, para detectar defectos de código.

Los diferentes tipos de representación se pueden combinar. Un artefacto representado con un AST se puede analizar con algoritmos de grafos, si se formaliza como un grafo o fórmulas lógicas. Un modelo se puede almacenar en una base de datos relacional para

Figura 2.5. Característica actividad y sus características

proporcionar una forma de acceso fácil y eficiente, mediante consultas a la base de datos.

2.5. Actividad

La característica actividad mostrada en la Fig. 2.5, representa un punto de variación entre los diferentes enfoques en la gestión de defectos. El proceso de gestión de los defectos se puede descomponer en diferentes tipos de actividades que pueden ser soportados en una determinada solución. En el diagrama de la Fig. 2.5, la característica se ha descompuesto en cinco tipos de actividades que se han encontrado en los trabajos inspeccionados. Para cada tipo de actividad se describe las técnicas utilizadas para soportarla, el grado de soporte de automatización alcanzado y el tipo de resultado que producen las diferentes soluciones. La característica sobre el grado de soporte de automatización se define en común para todas las actividades. En cambio, las características de técnica aplicada y resultados obtenidos, son específicas de cada actividad.

El grado de soporte de automatización de una actividad refleja la madurez de la aproximación estudiada. En la Fig. 2.6, se muestran los distintos niveles de automatización tratados, los cuáles se corresponden con una simplificación de los definidos por Sheridan [She00]:

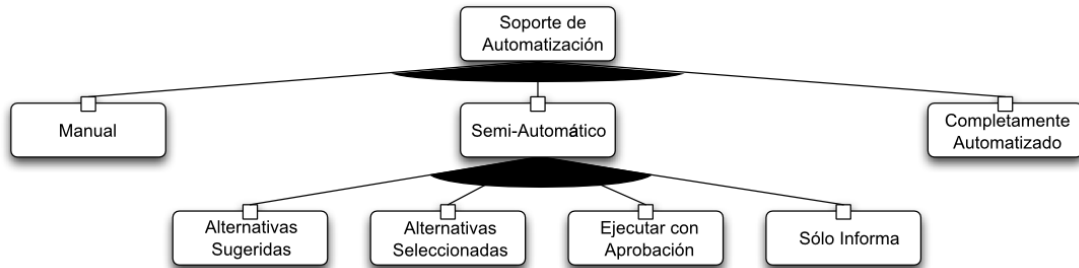
Manual: La actividad se realiza de forma manual.

Alternativas Sugeridas: La herramienta puede ejecutar la actividad automáticamente y sugiere opciones o alternativas al usuario. El usuario es el responsable de seleccionar y aplicar las sugerencias manualmente.

Alternativas Seleccionadas La herramienta sugiere y selecciona las tareas alternativas que se pueden realizar. El usuario es el responsable de confirmar la selección.

Ejecutar con Aprobación: La herramienta presenta al usuario la actividad que se va a ejecutar, pero requiere de su permiso. El usuario sólo puede elegir aplicar la actividad como un todo, o cancelarla.

Figura 2.6. Grado de automatización soportada en una actividad de gestión de defectos de diseño



Sólo Informa: La herramienta decide y ejecuta la actividad sin preguntar al usuario, pero informa al usuario del proceso.

Completamente Automatizado: La herramienta realiza la actividad de manera completamente automatizada, sin informar al usuario que está sucediendo.

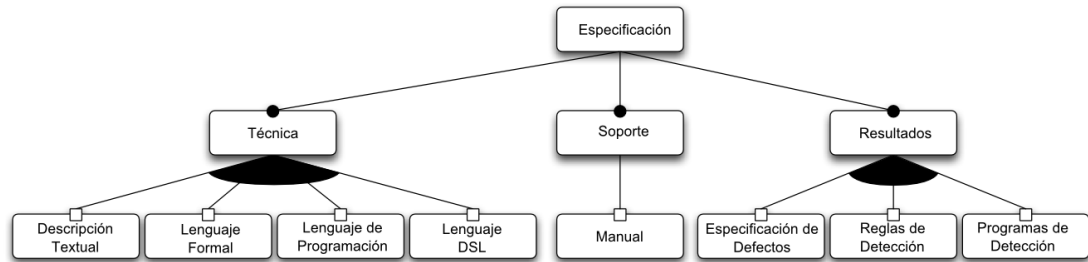
Cualquier gestión de defectos de diseño debe producir algún resultado cuando se aplica sobre un artefacto. Para comparar diferentes aproximaciones, hemos considerado necesario el tipo de resultado obtenido, o que puede ser obtenido, en cada actividad. Analizar una aproximación o una herramienta bajo esta dimensión es importante para determinar si su utilidad resuelve un problema particular, o si puede ser combinada con otro enfoque o tipo de herramientas.

En las siguientes subsecciones se muestra con más detalle el estado del arte de las distintas actividades (ver Fig. 2.5) relacionadas con las gestión de defectos de diseño.

2.5.1. Especificación

Las soluciones que implementan la actividad de especificación proporcionan a los desarrolladores el soporte necesario para extender o adaptarlo a sus necesidades particulares especificando nuevos defectos o modificando los ya existentes.

Técnica. Muchos de las soluciones inspeccionadas que soportan la actividad de especificación incluyen una descripción, al menos textual, del defecto de diseño para detectarlo o corregirlo. Típicamente, la técnica utilizada por las herramientas para especificar el defecto de diseño es a través del uso de reglas (posiblemente personalizadas) expresadas en algún tipo de: lenguaje formal (ej. OCL, SQL, XPath, fórmulas lógicas), lenguaje de programación (ej. Java) o lenguaje específico de dominio ([MGL⁺09]). Las especificaciones de los defectos definidas en [FBB⁺99], incluyen una descripción textual de guías generales para corregirlos. Wake [Wak03] especifica los defectos y sus correcciones en forma de plantillas. En el libro de antipatrones [BMMM98] proporciona una especificación mediante ejemplos narrados.

Figura 2.7. Resumen de la situación actual de la actividad de especificación.

Automatización. Varias aproximaciones o herramientas permiten describir reglas de detección y corrección como: PMD [PMD09], Eclipse’s Metrics plugin [Ecl], RefactorIt [Aqr02], iPlasma [Gro, LM06], Decor [Tea09] [Dur07]. Aunque esta actividad es intrínsecamente manual. La capacidad de definir o afinar las reglas de detección es común a la mayoría de los enfoques que tratan con defectos basados en avisos de métricas.

SemmlCode [Sem07] proporciona una interfaz bastante versátil para especificar defectos, a través de su lenguaje de consulta personalizado que permite escribir consultas complejas para detectar defectos de diseño.

Resultado. La actividad de especificación puede producir los siguientes tipos de resultados: una especificación puramente descriptiva y no automatizable, alguna clase de regla de detección que se pueda automatizar, o incluso un programa de detección ejecutable.

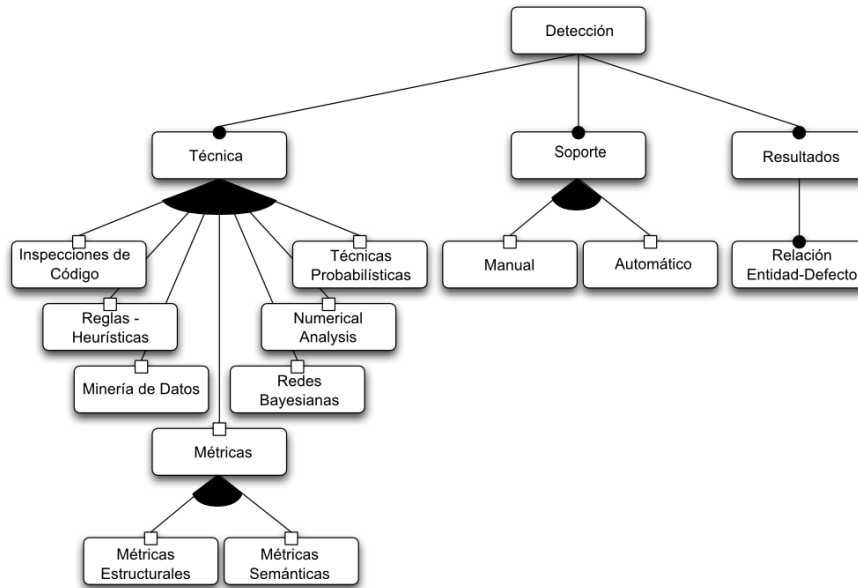
La Fig. 2.7 resume las técnicas, soporte de automatización y resultados de la actividad de especificación.

2.5.2. Detección

La mayoría de las soluciones existentes en la gestión de los defectos de diseño se centran en la actividad de detección.

Técnica. Uno de los enfoques para detectar defectos de diseño es a través de inspecciones manuales de código [TSFB99]. Muchas soluciones se basan en el uso de métricas, la gran mayoría de éstas se basan en métricas estructurales [CK91], como son [BEG⁺06, LM06, SLT06], pero algunas más recientes tienen en cuenta métricas semánticas [DLOV08]. Las métricas estructurales se corresponden con métricas derivadas de aspectos sintácticos del código orientado a objetos, como el análisis de las relaciones entre métodos y atributos de una clase.

Como parte de las métricas definidas por Chidamber and Kemerer [CK91] se encuentran: la profundidad del árbol de herencia (DIT), la carencia de cohesión (LCOM) y el acoplamiento entre objetos (CBO), que son ejemplos típicos de métricas estructurales.

Figura 2.8. Resumen de la actual situación de la actividad de detección

Las métricas semánticas se basan en el análisis de la información semántica embebida en el código, como los comentarios e identificadores [ED00]. El conocimiento basado en la comprensión del programa, y técnicas de procesamiento del lenguaje se utilizan para calcular estas métricas. Por ejemplo, la métrica semántica de cohesión de una clase LORM (Logical Relatedness of Methods) [ED00], mide la relación conceptual de los métodos de una clase determinada por la comprensión de los métodos de la clase que son representados por una red semántica de grafos conceptuales.

Otras soluciones utilizan reglas o heurísticas de conocimiento para detectar defectos de diseño [Ciu99, KRW07, LM06, MGD10]. Algunos enfoques recurren a técnicas más avanzadas procedentes del campo de la inteligencia artificial, como el uso de técnicas de minería de datos [XS04]; o del campo de la probabilidad como redes bayesianas [KVG09]; o del análisis numérico como B-Splines [OKAG10]. Aunque estas técnicas pueden ser insuficientes para detectar algunos defectos como *Shotgun Surgery* y *Divergent Change* [FBB⁺99], donde se deben considerar las probabilidades de propagación de un cambio en el diseño cuando cambia el artefacto. Rao et al. [RR08] proponen un método cuantitativo para detectar estos defectos con una matriz de probabilidades de propagación del cambio.

Automatización. Algunas de las propuestas son teóricas, con el objetivo de obtener una comprensión científica de las dificultades intrínsecas involucradas en la detección del defecto. Otros enfoques son completamente manuales, como el uso de técnicas de revisión y lectura para encontrar defectos [TSFB99]. La mayor parte de las aproximaciones inspeccionadas, además, proporcionan una herramienta de soporte como es el caso de [Chi02, Gro, MLC05b, MGD10, Sli05, Tri08, TCC08, WP05].

Resultado. En todas las soluciones inspeccionadas, por ejemplo en InCode e InFusion [Int08], la actividad de detección produce una lista de relaciones entre defectos y entidades. Estos resultados se presentan con diferentes medios, textuales o gráficos.

La Fig. 2.8 resume las técnicas, soporte de automatización y resultados para la actividad de detección.

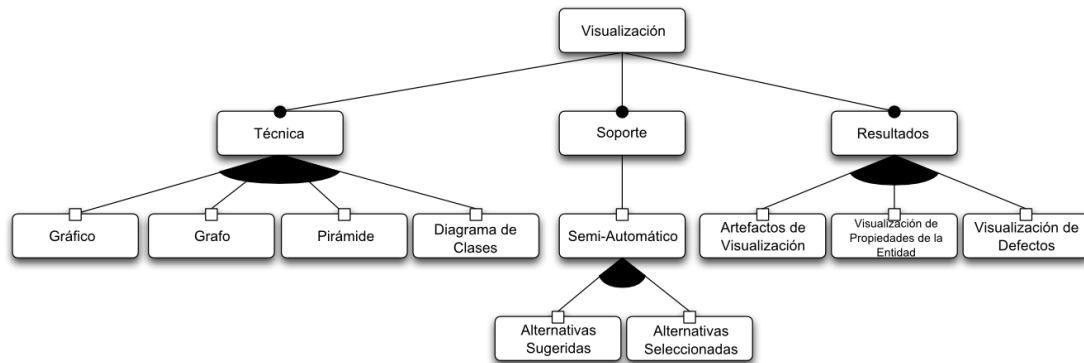
2.5.3. Visualización

La actividad de visualización produce algunos tipos de representaciones gráficas del artefacto objetivo, permitiendo identificar fácilmente alguna de sus propiedades. La mayor parte de las propuestas presentan la detección de los defectos de una forma gráfica, además la herramienta de visualización puede proporcionar otro tipo de información. La visualización puede ayudar a decidir a los desarrolladores cuál es la mejor modificación para eliminar un determinado defecto, o se puede usar para analizar la evolución de un sistema o explicar las causas y consecuencias de los defectos.

Técnica. El tipo de técnica de visualización utilizada depende principalmente del tipo de información que se quiere visualizar. Si la información es esencialmente una tabla de una hoja de cálculo, se puede visualizar como diagramas de sectores, de barras, de polígonos de frecuencia, etc. Si la información es un grafo, son necesarias técnicas de visualización de grafos y algoritmos de visualización gráficos más o menos sofisticados. Por ejemplo, esto ocurre si se quiere representar (parte de) la estructura de software, como las relaciones de dependencia [Con99, Tes08]. Algunas soluciones [LM06] definen nuevas técnicas de visualización, como la pirámide descriptiva (*Overview Pyramid*), la cuál está basada en métricas y sirve para describir y caracterizar la estructura de un sistema orientado a objetos cuantificando tamaño, complejidad, acoplamiento y uso de la herencia.

Automatización. La actividad de visualización generalmente requiere de intervención humana, ya sea durante la construcción de la representación visual (por ejemplo, seleccionando las áreas de interés o eligiendo la representación más apropiada o el algoritmo de diseño gráfico) o durante su uso. La visualización es estática si sólo se puede ver la representación gráfica [Con99, Tes08]. Por otro lado, la visualización es dinámica si los elementos gráficos se relacionan de alguna forma con el artefacto objetivo del estudio (por ejemplo, InCode e InFusion [Int08]). Este dinamismo mejora la velocidad en la detección y corrección de defectos.

Resultado. Muchas de las propuestas inspeccionadas soportan alguna clase de visualización, produciendo diferentes tipos de diagramas que ofrecen diferentes tipos de artefactos de visualización para guiar la comprensión del artefacto objetivo. Por ejemplo, en DECOR, los sistemas están representados como diagramas de clases y las clases con defectos están destacadas con el color rojo. Otros tipos de visualizaciones que ayudan en la gestión de los defectos de diseño son: la visualización de propiedades de las entidades, visualizaciones de los defectos de diseño [vM02], la pirámide descriptiva, las vistas polimétricas [LM06] o grafo de dependencias [Con99].

Figura 2.9. Resumen de la situación actual de la actividad de visualización

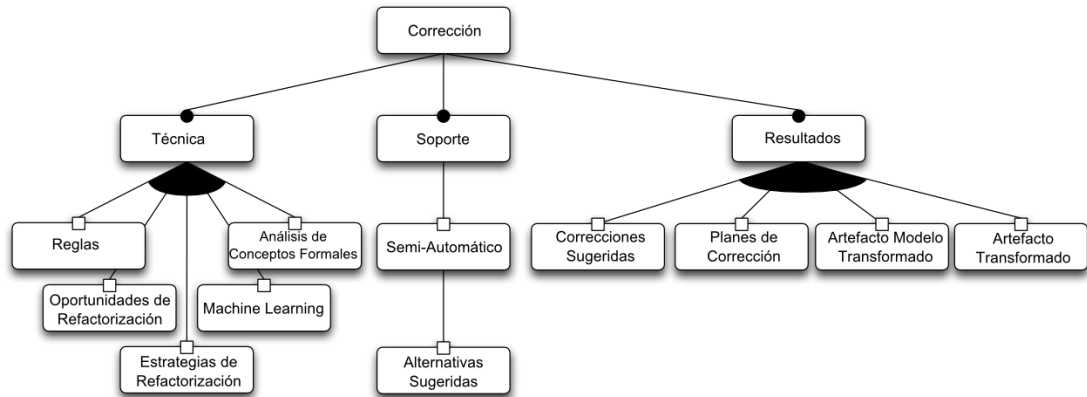
La Fig. 2.9 resume las técnicas, soporte de automatización y resultados para la actividad de visualización.

2.5.4. Corrección

Los enfoques que soportan la actividad de corrección de defectos de diseño, proporcionan una forma de (sugerir como) modificar el artefacto objetivo, para eliminar los defectos y mejorar su diseño. En la inspección realizada hemos encontrado menos soluciones que soportan una corrección automática (parcialmente), ya que esta actividad está menos madura que las actividades de detección y visualización.

Técnica. Durante la inspección hemos encontrado principalmente cinco técnicas diferentes para corregir defectos. La primera técnica es aplicar reglas sobre los defectos de diseño que se han detectado previamente [MGDL10, TM03, TSG04]. Estos enfoques tienen la ventaja de proporcionar un proceso comprensivo tanto para la detección como para la corrección. La segunda categoría sugiere correcciones, llamadas oportunidades de refactorización, basándose sólo en valores de métricas o la presencia de ciertos patrones, sin explicitar la identificación de los defectos de diseño [BCT07, DLOV08, GAA01, SGM00, SSL01, SS07, TK04]. La tercera técnica, estrategias de refactorización, son especificaciones automatizables de secuencias de refactorización complejas dirigidas a un objetivo particular, como la corrección de defecto de diseño [Pér11]. La cuarta técnica se basa en ideas que proceden del campo del aprendizaje automático (*machine learning*), donde se explotan técnicas como los algoritmos genéticos y otros tipos de aprendizaje [BCT07, BAMN06]. La quinta técnica es el análisis de conceptos formales (FCA) [GW99] y ha sido investigada intensamente para reestructurar la jerarquía de clases [ADN05, SLMM99, ST00] y clases afectadas por defectos [MHVG08].

Automatización. Varias herramientas soportan de manera semiautomática la acti-

Figura 2.10. Resumen de la situación actual de la actividad de corrección

vidad de corrección de defectos [BEG⁺06, TSG04, TCC08], aunque todas necesitan alguna interacción adicional con el usuario.

Resultado. La corrección de defectos de diseño puede producir distintos tipos de resultados que dependen del grado de automatización de cada herramienta particular. Algunas herramientas proporcionan sólo sugerencias de corrección, otras proporcionan planes de corrección o especificaciones de secuencias de transformación necesarias para mejorar el diseño del artefacto objetivo. Por ejemplo, Trifu et al. [TSG04] proponen estrategias de corrección mapeando defectos de diseño con posibles soluciones. Sin embargo, una solución es un ejemplo de cómo el programa *debería haberse implementado* para evitar un defecto, no un lista de pasos a seguir por un desarrollador de software para corregir el defecto.

Algunas herramientas pueden aplicar los cambios en la representación interna del artefacto, produciendo el artefacto transformado. Las herramientas completamente automatizadas podrían operar directamente sobre el artefacto, generando el artefacto transformado. Un ejemplo es prototipo citado en [Pér11].

La Fig. 2.10 resume las técnicas, soporte de automatización y resultados de la actividad de corrección.

2.5.5. Análisis de impacto

La actividad de análisis de impacto se refiere a la capacidad de computar el impacto del cambio de un defecto de diseño [vM02, VDDR07] o las acciones realizadas para eliminarlo [FTC07, LWN07b, TSG04].

Técnica. Aproximaciones basadas en modelos de calidad ofrecen esta característica [BAMN06, LM06, RSS⁺04, SGM00, TK04]. En [TK04], Tahvildari et al. presentan una propuesta basada en modelos de objetivos (*soft-goal models*). Con este tipo de modelos

definen los efectos de los defectos de diseño sobre métricas y factores de calidad del sistema de manera que esta información se puede usar para automatizar herramientas de detección o de corrección. Marinescu introduce en [Mar02a] cómo los modelos de calidad se pueden utilizar para estimar el impacto de un defecto de diseño.

Deligiannis et al. [DSA⁺04] presentan un experimento controlado sobre el impacto de los defectos de diseño, en el cual estudian a 20 sujetos para evaluar el impacto de clases con el defecto *God Class* respecto de la facilidad de mantenimiento y la facilidad de comprensión de los sistemas orientados a objetos. Los resultados del estudio muestran que el antipatrón *Blob* afecta a la evolución de las estructuras de diseño. Otros enfoques similares, basados en experimentos controlados estudian el impacto de los defectos de diseño en factores de calidad del software como la facilidad de comprensión [DDV⁺06] y la facilidad de mantenimiento [OCBZ09]. Algunos enfoques utilizan modelos estadísticos para investigar la relación entre los defectos de diseño y la probabilidad de error en la clase [LS07] o con su propensión al cambio [KPG09].

Trabajos recientes han estudiado el impacto de los defectos de diseño en la evolución del software analizando varias versiones de sistemas software [OCBZ09, VKMG09]. Estas aproximaciones identifican principalmente patrones de evolución de defectos, que se pueden usar para explicar el impacto de los defectos sobre el resto del sistema. Por ejemplo, Olbrich et al. [OCBZ09] analizan el histórico de datos de varios años de desarrollo de dos proyectos de gran escala de código abierto. El estudio concluye que las clases con los defectos *God Class* y *Shotgun Surgery* tienen una mayor frecuencia de cambio que otras clases, pudiendo necesitar más mantenimiento que clases no afectadas por estos defectos.

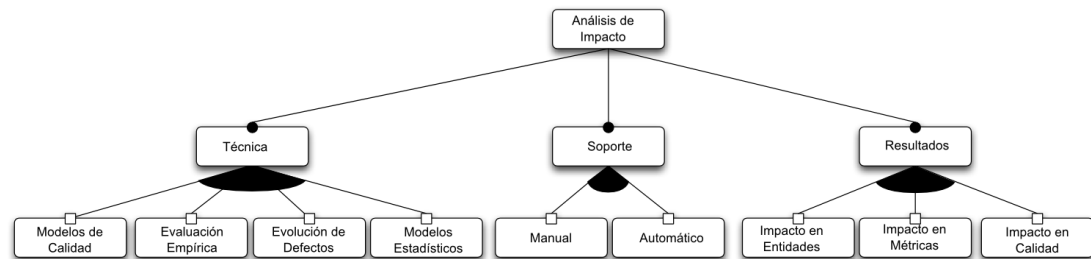
Automatización. Muchas de las soluciones estudiadas evalúan el impacto de los defectos de diseño y lo validan con experimentos controlados [DSRS03, DSA⁺04, DDV⁺06]. Algunas soluciones integran esta actividad de manera totalmente automática [TK04] para asistir el proceso de detección y corrección.

Resultados. Los resultados más comunes en las propuestas relacionadas con el análisis de impacto son listas de entidades afectadas por un defecto de diseño (impacto de entidades), o el efecto que un defecto, o su eliminación, tienen sobre el valor de una métrica (impacto en métricas) [DSRS03] o sobre un factor de calidad (impacto en calidad) [DSA⁺04, DDV⁺06].

La Fig. 2.11 resume las técnicas, soporte de automatización y resultados de la actividad análisis de impacto.

2.6. Caracterización de herramientas: inCode y JDeodorant

En esta sección se aplica la taxonomía a las herramientas de gestión de defectos, inCode [Int08] y JDeodorant [JDe11], que hemos utilizado para obtener predicciones iniciales de entidades con defectos. Esta predicción inicial es necesaria para aplicar el proceso definido en la Sec. 4.1. Con ellas se forman los conjuntos de datos de entrenamiento uti-

Figura 2.11. Resumen de la situación actual de análisis de impacto

lizados en la actividad de experimentación para obtener las heurísticas de identificación (árbol de decisión) de cada defecto (ver Fig. 4.2).

Tanto inCode como JDeodorant son plugins de Eclipse que ayudan al desarrollador o auditor a comprender, evaluar y mejorar la calidad del diseño de un proyecto. Están totalmente integradas en el entorno de desarrollo, asisten al usuario en las tareas de mantenimiento de código Java y le permiten detectar y corregir problemas de diseño de forma continua y ágil.

En inCode la tarea de detección de defectos de diseño se aplica sobre los defectos (*bad smells*) conocidos como: la duplicación de código (*Duplicated code*), las clases que rompen la encapsulación (*Data class*, *God class*), o métodos que se encuentran en una clase equivocada (*Feature envy*). La detección se basa en métricas orientadas a objetos.

Por su parte, JDeodorant identifica cuatro *bad smells*: *Feature envy*, *Type checking* vs. *Switch statement*, *Long method* y *God class*.

Al trabajar ambas herramientas con defectos comunes nos ayudará a elaborar conjuntos de datos más precisos. inCode, además, calcula estadísticos de agrupamiento con valores de mediciones aplicados a paquetes, subsistemas y sistemas. Son accesibles desde la opción *overview* donde además aparecen mediciones relacionadas con la herencia. En la Tabla 2.2 se muestra un resumen de lo comentado y representado en la taxonomía como la característica *defecto de diseño*.

En ninguna de las dos se admite la comparación de versiones. El tipo de representación interna con la que trabaja JDeodorant es AST (*Abstract Syntax Tree*). En inCode se desconoce el tipo de representación para el análisis.

En la Tabla 2.3 se muestra un resumen de lo comentado y representado en la taxonomía como la característica *artefacto*. Un guión dentro del contenido de las celdas indica que el valor es desconocido.

inCode presenta tres tipos de actividades: detección, visualización y corrección, por otro lado JDeodorant presenta: detección, corrección y análisis de impacto. En ambas herramientas, la actividad de detección se basa en métricas. El usuario interactúa con la herramienta para seleccionar el defecto y conjunto de entidades. Además de esta interacción, inCode presenta una detección en segundo plano con la que, de manera

Tabla 2.2. Característica defecto de diseño en Incode y JDeodorant

Herramienta	Tipo de defecto	Nivel	Ámbito					Propiedades
			Sistema	Subsistema	Paquete	Clase	Método	
inCode	Bad Smell Disharmonies Metrics warning	Bajo Alto	Si	Si	Si	Si	Si	Medibles Estructurales
JDeodorant	Bad Smell Disharmonies	Bajo Alto	No	No	No	Si	Si	Medibles Estructurales

Tabla 2.3. Característica artefacto en inCode y JDeodorant

Herramienta	Entorno de trabajo	Tipo artefacto	Versiones	Tipo de representación
inCode	Plugin Eclipse	código fuente (Java)	No	-
JDeodorant	Plugin Eclipse	código fuente (Java)	No	AST

totalmente automática, va señalando los posibles problemas de diseño en el código con el uso de marcadores rojos junto a la línea de código.

En ambas herramientas, la actividad de corrección aplica técnicas de refactorización indicadas textualmente y se integran con el conjunto refactorizaciones incluidas en Eclipse. La interacción de las herramientas se hace desde la tabla de entidades que tienen defecto, a partir de una entidad con defecto se sugiere textualmente una refactorización que puede ser ejecutada.

inCode presenta una fuerte actividad de representaciones gráficas proponiendo cuatro representaciones gráficas interactivas para capturar aspectos esenciales de las entidades del sistema a partir de vistas polimétricas: complejidad del sistema, esquema de clase (*class blueprint*), interacción de métodos y uso de atributos.

Como actividad característica de JDeodorant se encuentra el análisis de impacto. Esta actividad sólo está disponible para *Feature Envy*. Existe un campo denominado *entity emplacement* que toma diferentes valores en función de los defectos detectados en las clases. Su interpretación sirve para modelar el impacto que puede provocar una refactorización de manera virtual. El usuario dispone de un criterio para aplicar las refactorizaciones más eficaces. JDeodorant muestra los candidatos a refactorización en orden ascendente con el fin de evidenciar cuál sería la solución más efectiva. En la Tabla

Tabla 2.4. Característica actividad en inCode y JDeodorant

Herramienta	Tipo de actividad	Técnica	Automatización	Tipo de Resultado	Formato de Salida
inCode	Detección	Basado en métricas	Interactiva, completamente automatizado	Relaciones entre entidades y defectos	Textual
	Visualización	Basada en visualización	Interactiva	Vistas polimétricas	Visual
	Corrección	Sugiere refactorizaciones	Interactiva	Acciones de refactorización	Modelos
JDeodorant	Detección	Basado en métricas	Interactiva	Relaciones entre entidades y defectos	Textual y numérico
	Corrección	Sugiere refactorizaciones	Interactiva	Acciones de refactorización	Textual y modelos
	Análisis de impacto	Basada en métricas	Interactiva	Basado en métricas	Numérica

2.4 se muestra un resumen de lo comentado y representado en la taxonomía como la característica *actividad*.

2.7. Caracterización de la tesis

En la tesis doctoral, los tipos de defectos de diseño tratados no están seleccionados por autores, sino porque puedan ser detectados utilizando medidas de código. La naturaleza de diseño de la entidad es un factor que no ha sido considerado explícitamente en ninguna de las propuestas de automatización analizadas. Para cubrir esta carencia, la principal aportación de este trabajo es considerar la naturaleza de diseño de la entidad a medir, entendida como estereotipos de clasificadores estándar de UML («interface», «test», «entity», «control», «util», «sin clasificar»).

Estos estereotipos UML proporcionan una fuente de información utilizada en algunas definiciones textuales de defectos de Fowler y Beck [FBB⁺99] para considerar casos

excepcionales. Por ejemplo, los métodos de las clases estereotipadas como «test» tienen la responsabilidad de probar la funcionalidad de otras clases provocando dependencias con ellas. Además las buenas prácticas de diseño de pruebas aconsejan que cada método pruebe sólo una funcionalidad. Si no disponemos de la información del estereotipo de la clase, que sirva para modelar este caso excepcional, entendemos que estos métodos, a pesar de estar bien diseñados, serán identificados como afectados por el defecto *Feature Envy*.

Los tipos de defectos que pueden estar afectados por la naturaleza de la entidad, pueden ser tanto de bajo como de alto nivel. Al igual que con los tipos, la naturaleza puede ser asignada a entidades de cualquier ámbito.

Hasta donde llega nuestro conocimiento no existen trabajos orientados a inferir el estereotipo de una entidad de código. Por ello, en este trabajo se incluirán, además de propiedades medibles, propiedades léxicas para inferir esta propiedad. Continuando con el ejemplo del estereotipo «test», se puede inferir este estereotipo en una entidad cuyo nombre cualificado contenga la cadena “test”.

Tal como se menciona al principio del capítulo, una de las aplicaciones de la taxonomía es caracterizar trabajos de investigación para mejorar métodos y técnicas, aplicadas a la investigación en la detección de defectos de diseño en orientación a objetos. En este sentido, hemos aplicado dicha taxonomía en nuestro propio trabajo, analizando las tres características de primer nivel: Defectos de diseño, Artefacto objetivo y Actividad. En la Tabla 2.5 se resume la característica defecto de diseño de la taxonomía presentada.

Tabla 2.5. Característica defecto de diseño en esta tesis

Herramienta	Tipo de defecto	Nivel	Ámbito					Propiedades
			Sistema	Subsistema	Paquete	Clase	Método	
Tesis	Bad Smell Disharmonies Antipatronos	Bajo Alto	Si	Si	Si	Si	Si	Medibles Léxicos

En esta tesis la característica artefacto se corresponde con un conjunto de medidas obtenidas sobre una única versión del código fuente (Java). El entorno de trabajo de los prototipos generados como prueba de concepto son extensiones de Eclipse o adaptaciones de extensiones ya existentes. El modelo de representación del artefacto utilizado en esta tesis para proponer un framework de medidas es un modelo de objetos, basado en el lenguaje MOON [LMC03].

En la Tabla 2.6 se muestra un resumen de la aplicación de la característica artefacto en esta tesis.

Esta tesis se centra en dos actividades: la especificación interactiva con el usuario y la

Tabla 2.6. Característica artefacto en esta tesis

Herramienta	Entorno de trabajo	Tipo artefacto	Versiones	Tipo de representación
Tesis	Plugin Eclipse	código fuente (Java)	No	Modelo Objetos

detección automática. En la detección teórica de defectos, un aspecto destacable, es la subjetividad asociada al proceso, en [FBB⁺99] se delega en la intuición del inspector como criterio final de decisión. Parte de esa subjetividad está asociada al contexto de la aplicación, biblioteca o framework que se esté inspeccionando, por tanto, las reglas de detección deberían adaptarse al contexto a partir de decisiones del inspector. Algunas propuestas estudiadas se enfrentan al problema de la subjetividad modificando los valores umbrales (absolutos vs. relativos) de sus heurísticas de detección. La realización de esta operación no es fácil de llevar a cabo por un usuario, en el sentido de que pueda evaluar la bondad del nuevo conjunto de entidades resultado de aplicar los nuevos umbrales.

En esta tesis se define una nueva técnica de especificación de defectos donde el usuario define el defecto indicando ejemplos de entidades que lo tienen. Posteriormente con técnicas de clasificación de minería de datos se obtendrán las heurísticas utilizadas para su detección. Por tanto, las reglas de detección se generan dinámicamente en función del contexto de trabajo que determina el usuario.

En la Tabla 2.7 se muestra un resumen de la aplicación de la característica actividad sobre esta tesis.

Tabla 2.7. Característica actividad en esta tesis

Herramienta	Tipo de actividad	Técnica	Automatización	Tipo de Resultado	Formato de Salida
Tesis	Especificación	Basada en un conjunto de medidas de entidades	Interactiva	Reglas de detección	
	Detección	Minería de datos	Automático	Relaciones entre entidades y defectos	Textual

PROCESO DE MEDICIÓN MODIFICADO Y SU VALIDACIÓN

Una de las aplicaciones del proceso de medición de software es la identificación de mediciones anómalas usando valores umbrales de diferentes métricas. Algunos autores realizan estudios sobre la determinación de valores umbrales. En algunos casos los valores son absolutos, es decir, se aplican los mismos a diferentes productos. En otros casos son relativos a cada producto concreto [LM06].

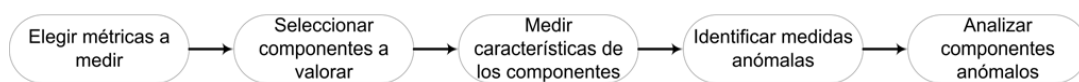
En el caso particular de la medición de código, se utilizan los mismos valores umbrales independientemente de la naturaleza del problema que resuelve la entidad a medir. En este capítulo se proponen las modificaciones al proceso de medición para incorporar esta nueva variable. Por tanto, el proceso debe incluir actividades relacionadas con la clasificación de entidades según su naturaleza. En este caso, las categorías de dicha clasificación son los estereotipos estándar UML («entity», «control», «interface», «util», «test», «exception»). Por eso hay que considerar en este trabajo el uso como sinónimos de intención de diseño, naturaleza de diseño y estereotipos estándar UML.

La validación de esta modificación del proceso de medición que proponemos se ha realizado mediante experimentación empírica, por medio de dos casos de estudio. El primero tiene como objetivo estudiar si las métricas de código se comportan igual en las entidades, independientemente de su naturaleza. Su objetivo es evaluar la utilidad del nuevo parámetro, naturaleza de diseño de la entidad, que hemos incorporado al proceso de medición. Este caso de estudio presenta una amenaza fuerte por la subjetividad del método de clasificación automática de entidades respecto a su naturaleza. Por eso se realizó el segundo caso de estudio, cuya finalidad es evaluar dicha amenaza, estudiando el grado de afinidad entre la clasificación realizada por humanos y un método de clasificación automático.

3.1. El proceso de medición

Desde la década de los 90, las métricas del software y el proceso de medición asociado, han captado la atención de la comunidad de la Ingeniería del Software como medio para cuantificar y controlar la calidad del software [FN99, IEE05, Pre05]. Según Sommerville en [Som05], la medición es una actividad que forma parte de un proceso (ver Fig.3.1), que consiste en asociar valores numéricos a atributos de productos o procesos de software.

Figura 3.1. Proceso de medición definido por Sommerville



Pero en la bibliografía también existen muchas críticas sobre el uso que se hace de las métricas del software [Mar02b]. Una de ellas es que los valores umbrales utilizados para identificar medidas anómalas, obtenidas a través de experimentos empíricos, están restringidos al contexto de medición, siendo limitada su utilización en otros contextos. Incluso valores umbrales recomendados extraídos de históricos de medidas de un mismo contexto, no son aplicables igualmente a todas las entidades de dicho contexto. Las preguntas concretas que nos planteamos a la hora de trabajar con valores umbrales para identificar componentes anómalos son:

- ¿Es correcto usar valores umbrales absolutos o relativos al producto?
- ¿Cuál es la influencia del tamaño y tipo de software [GHJV95, FSJ99] (framework de dominio, framework de aplicación y bibliotecas) en los valores umbrales?
- ¿Es adecuado el uso de los mismos valores umbrales en diferentes versiones del mismo producto?

Para contestar a estas preguntas, nos planteamos recoger mediciones de código en diferentes productos software. Los resultados fueron discutidos en un taller especializado de medición (QAOOSE [CLM06]) y se presentan en las siguientes subsecciones.

3.1.1. Estudio de los valores umbrales en distintos productos

Para estudiar la influencia del tamaño y de los distintos tipos de producto en los valores umbrales de las métricas, se compararon seis productos de diferentes tamaños, muchos de ellos con versiones estables, usadas durante períodos largos de tiempo.

Todos los productos seleccionados están escritos en Java y son de código abierto. Los dominios de aplicación donde se pueden utilizar son variados: automatización y cobertura de pruebas, diseño de componentes gráficos, desarrollo de aplicaciones web, generación de gráficos estadísticos. Además se han seleccionado diferentes tipos de software:

bibliotecas de clases, framework de dominio y de aplicación [FSJ99]. Los productos seleccionados y caracterizados son los siguientes:

- jfreechart-1.0.0.pre2 (629 clases, gráficos estadísticos, biblioteca de clases)
- jhotdraw-6.0b1 (496 clases, diseño de componentes gráficos, framework de dominio)
- struts-1.2.8 (273 clases, desarrollo de aplicaciones web, framework de dominio)
- jcoverage-1.0.5 (90 clases, cobertura de pruebas, biblioteca de clases)
- easymock-1.0.5 (47 clases, automatización de pruebas, framework de aplicación)
- junit-3.8.1 (46 clases, automatización de pruebas, framework de aplicación)

En el estudio se usó Eclipse 3.1 y el plugin Metrics 1.3.6 para poder recoger las medidas sobre clases. Las medidas elegidas están relacionadas con tamaño, complejidad, cohesión y herencia [CK94, LK94]:

- NOF número de campos
- NOM número de métodos
- WMC métodos ponderados por clase
- LCOM carencia de cohesión de los métodos
- DIT profundidad del árbol de herencia
- NOC número de hijos
- SIX índice de especialización
- NORM número de métodos redefinidos

Para cada métrica, se obtienen los siguientes valores estadísticos: media, media acotada (eliminado el 15 % de los valores extremos), desviación estándar, cuartil inferior (Q1), mediana (Q2), cuartil superior (Q3), mínimo y máximo.

A partir de los datos descriptivos presentados en Fig. 3.2, se observa que:

- Las distribuciones no son simétricas, existen diferencias entre la media y la mediana, además la mediana está próxima a Q3. En muchos de los casos se obtienen distribuciones con asimetría positiva (la cola de la distribución está a la derecha).
- Las diferencias entre los mínimos y los máximos son grandes. Esto sugiere que las medidas están dispersas y que pueden tener diferentes valores umbrales.
- El tamaño del producto (número de clases) está correlacionado con algunas métrica como: NOF, NOM y WMC. Por contra, métricas como LCOM y DIT muestran una variación escasa entre los diferentes productos.

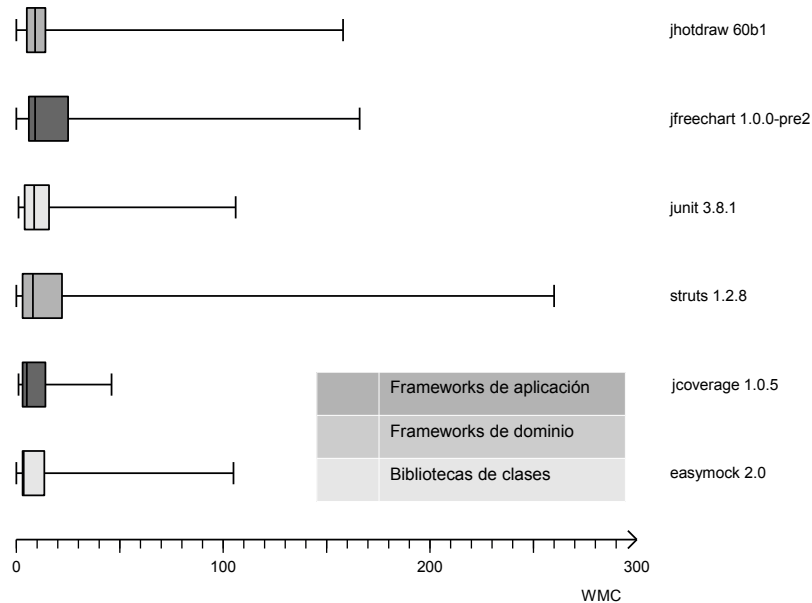
En la Fig. 3.3 se muestra un gráfico de cajas definido sobre la métrica WMC para cada uno de los productos. Los productos están ordenados de arriba a abajo, según el valor de su mediana. Se puede apreciar que hay diferencia entre las distribuciones de

Figura 3.2. Resultados descriptivos globales

	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean JFreeChart 1.0.0-pre2	2,40	10,08	22,98	0,21	2,55	0,36	0,16	0,69
Bounded mean (15%)	1,41	7,45	15,87	0,17	2,47	0,04	0,08	0,46
Q3	3,00	11,00	25,00	0,50	3,00	0,00	0,14	1,00
Q2 Median	1,00	5,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	3,00	6,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	5,05	15,01	38,82	0,32	1,14	1,48	0,37	1,23
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	48,00	166,00	490,00	1,00	7,00	16,00	3,20	9,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Junit-3.8.1	2,17	8,13	15,70	0,21	2,70	0,28	0,18	0,35
Bounded mean (15%)	1,50	6,53	12,33	0,18	2,58	0,15	0,09	0,28
Q3	2,00	9,75	15,75	0,50	3,75	0,00	0,12	1,00
Q2 Median	1,00	4,50	8,00	0,00	2,00	0,00	0,00	0,00
Q1	0,00	2,00	4,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	3,59	10,35	20,42	0,33	1,84	0,72	0,45	0,60
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	18,00	62,00	106,00	0,91	6,00	3,00	2,00	3,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Jcoverage-1.0.5	1,49	4,35	9,56	0,24	1,78	0,39	0,81	0,28
Bounded mean (15%)	1,23	3,70	8,17	0,20	1,62	0,19	0,10	0,21
Q3	2,00	5,00	14,00	0,50	2,00	0,00	0,00	0,00
Q2 Median	1,00	3,00	5,00	0,00	1,00	0,00	0,00	0,00
Q1	0,00	2,00	3,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	1,87	4,46	9,59	0,34	1,05	0,96	0,37	0,52
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	7,00	25,00	46,00	1,00	5,00	4,00	1,67	2,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean easymock-2.0	1,41	5,83	12,54	0,15	1,24	0,09	0,12	0,33
Bounded mean (15%)	1,24	4,13	8,32	0,11	1,08	0,00	0,02	0,16
Q3	2,00	5,00	13,50	0,33	1,00	0,00	0,00	0,00
Q2 Median	1,00	3,00	3,50	0,00	1,00	0,00	0,00	0,00
Q1	1,00	3,00	3,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	1,34	7,51	19,25	0,24	0,67	0,46	0,41	0,73
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	6,00	38,00	105,00	0,85	4,00	3,00	2,00	3,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean struts-1.2.8	2,91	8,60	18,84	0,28	2,59	0,46	0,51	0,96
Bounded mean (15%)	2,09	6,66	13,21	0,25	2,45	0,24	0,33	0,67
Q3	4,00	11,00	22,00	0,67	4,00	1,00	0,60	1,00
Q2 Median	2,00	4,00	8,00	0,00	2,00	0,00	0,00	0,00
Q1	0,00	2,00	3,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	4,56	11,02	29,13	0,36	1,48	1,13	0,95	2,04
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	40,00	82,00	260,00	0,98	7,00	10,00	5,00	28,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean JHotDraw60b1	1,40	9,51	13,36	0,16	2,84	0,57	0,31	0,73
Bounded mean (15%)	1,09	7,72	10,31	0,11	2,68	0,07	0,16	0,38
Q3	2,00	11,00	14,00	0,00	4,00	0,00	0,32	1,00
Q2 Median	1,00	7,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	4,00	5,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	1,86	10,40	16,76	0,30	1,49	3,84	0,74	1,70
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	19,00	90,00	158,00	1,50	9,00	71,00	8,00	19,00

los diferentes productos, lo que confirma la sospecha de que el producto puede afectar a los valores umbrales de las métricas. Respecto al tipo de software (ver el relleno de las cajas en la Fig. 3.3), se aprecia que los valores máximo y mínimo de productos del mismo tipo de software son muy diferentes.

Una primera conclusión, a la vista de estos resultados, es que no se puede hablar en términos absolutos de valores umbrales que identifican medidas anómalas de entidades de código.

Figura 3.3. Gráficos de cajas de WMC de diferentes productos y tipos de software

3.1.2. Estudio de los valores umbrales en la evolución del producto

Puesto que sospechamos que no se puede disponer de valores umbrales absolutos, para detectar medidas anómalas, proponemos una primera hipótesis:

La variación del tamaño del producto, en sus sucesivas versiones, afecta a las medidas de entidades de código.

Para verificar esta hipótesis se define un nuevo estudio empírico en el que los objetos a observar serán un conjunto de versiones de los productos. En concreto, se obtienen versiones de tres productos: JFreechart, JHotDraw y JUnit. Para analizar la evolución de los productos, se considera un rango de 4 ó 5 años entre las versiones. A continuación se muestra el número de clases de cada versión y un análisis descriptivo de los datos.

Para JFreeChart se analizan las siguientes versiones:

- jfreechart-1.0.1 (691 clases, 2006-01-27)
- jfreechart-1.0.0-pre2 (629 clases, 2005-03-10)
- jfreechart-0.9.21 (570 clases, 2004-09-10)
- jfreechart-0.9.7 (492 clases, 2003-04-17)
- jfreechart-0.9.4 (326 clases, 2002-10-18)

En el caso de JHotDraw, la versión, el número de clases y las fechas son:

- jhotdraw-6.0b1 (497 clases, 2004-02-01)

Figura 3.4. Análisis descriptivo de JFreeChart

	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-1.0.1	2,22	9,94	22,42	0,19	2,53	0,33	0,16	0,69
Bounded mean (15%)	1,27	7,27	15,25	0,15	2,46	0,03	0,09	0,48
Q3	2,00	11,00	23,00	0,40	3,00	0,00	0,17	1,00
Q2 Median	1,00	5,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	4,00	7,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	4,86	15,18	39,39	0,31	1,12	1,41	0,35	1,18
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	46,00	173,00	513,00	1,00	7,00	14,00	3,33	8,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-1.0.0-pre2	2,40	10,08	22,98	0,21	2,55	0,36	0,16	0,69
Bounded mean (15%)	1,41	7,45	15,87	0,17	2,47	0,04	0,08	0,46
Q3	3,00	11,00	25,00	0,50	3,00	0,00	0,14	1,00
Q2 Median	1,00	5,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	3,00	6,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	5,05	15,01	38,82	0,32	1,14	1,48	0,37	1,23
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	48,00	166,00	490,00	1,00	7,00	16,00	3,20	9,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-0.9.21	2,38	9,99	22,47	0,21	2,52	0,36	0,16	0,69
Bounded mean (15%)	1,41	7,33	15,44	0,17	2,45	0,05	0,08	0,44
Q3	2,00	12,75	26,00	0,50	3,00	0,00	0,16	1,00
Q2 Median	1,00	5,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	3,00	6,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	4,93	15,20	38,66	0,32	1,12	1,47	0,37	1,20
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	47,00	155,00	473,00	0,96	7,00	16,00	3,00	8,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-0.9.7	2,14	7,03	15,63	0,20	3,21	0,31	0,17	0,49
Bounded mean (15%)	1,22	5,06	11,08	0,15	3,07	0,04	0,07	0,28
Q3	2,00	9,00	19,00	0,50	4,00	0,00	0,09	1,00
Q2 Median	0,00	3,00	6,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	1,00	3,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	4,55	10,65	24,45	0,32	1,97	1,34	0,44	1,02
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	39,00	87,00	203,00	1,00	7,00	15,00	3,00	7,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-0.9.4	2,69	7,77	18,00	0,26	3,02	0,40	0,27	0,61
Bounded mean (15%)	1,71	6,13	13,51	0,22	2,85	0,08	0,11	0,32
Q3	3,00	11,00	24,00	0,62	4,00	0,00	0,16	1,00
Q2 Median	1,00	4,00	8,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	1,00	3,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	4,87	9,70	25,11	0,35	1,95	1,49	0,70	1,34
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	39,00	60,00	195,00	1,00	7,00	16,00	6,00	8,00

- jhotdraw-5.4b2 (478 clases, 2004-01-31)
- jhotdraw-5.3 (208 clases, 2002-01-20)
- jhotdraw-5.2 (149 clases, 2001-02-18)

En el caso de JUnit, la versión y el número de clases ¹ son:

- junit-3.8.1 (47 clases)
- junit-3.2 (32 clases)
- junit-2.1 (19 clases)

Para cada uno de los productos se realizan las mediciones de la métricas mencionadas calculando la media, media acotada (eliminado el 15% de los valores extremos), des-

¹las fechas de la versión no están disponibles

Figura 3.5. Análisis descriptivo de JHotDraw

	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean JHotDraw60b1	1,40	9,51	13,36	0,16	2,84	0,57	0,31	0,73
Bounded mean (15%)	1,09	7,72	10,31	0,11	2,68	0,07	0,16	0,38
Q3	2,00	11,00	14,00	0,00	4,00	0,00	0,32	1,00
Q2 Median	1,00	7,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	4,00	5,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	1,86	10,40	16,76	0,30	1,49	3,84	0,74	1,70
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	19,00	90,00	158,00	1,50	9,00	71,00	8,00	19,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jhotdraw54b1	1,41	9,67	13,89	0,16	2,90	0,58	0,32	0,73
Bounded mean (15%)	1,12	7,85	10,80	0,11	2,75	0,08	0,17	0,39
Q3	2,00	11,00	15,00	0,00	4,00	0,00	0,33	1,00
Q2 Median	1,00	7,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	1,00	4,00	6,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	1,81	10,33	16,88	0,30	1,48	3,89	0,75	1,71
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	16,00	88,00	148,00	1,50	9,00	71,00	8,00	19,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jhotdraw53	1,83	9,12	15,51	0,27	2,65	0,86	0,51	1,21
Bounded mean (15%)	1,46	7,07	11,60	0,23	2,43	0,20	0,41	0,97
Q3	3,00	10,00	18,00	0,63	3,00	0,00	0,75	2,00
Q2 Median	1,50	6,50	12,50	0,00	1,00	0,00	0,00	0,00
Q1	0,00	3,00	4,75	0,00	2,00	0,00	0,00	0,00
Standard Deviation	2,38	10,87	20,54	0,35	1,66	3,53	0,66	1,66
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	17,00	72,00	146,00	1,50	8,00	40,00	3,00	12,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jhotdraw52	1,83	8,30	13,53	0,26	2,81	0,68	0,56	1,28
Bounded mean (15%)	1,52	6,37	10,25	0,23	2,60	0,24	0,48	1,06
Q3	3,00	10,00	15,25	0,60	3,00	0,00	1,00	2,00
Q2 Median	1,00	5,00	8,00	0,00	2,00	0,00	0,28	1,00
Q1	0,00	3,00	4,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	2,19	9,82	16,84	0,33	1,70	1,91	0,66	1,69
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	14,00	61,00	108,00	1,50	8,00	12,00	3,11	12,00

Figura 3.6. Análisis descriptivo de JUnit

	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Junit 3.8.1	2,17	8,13	15,70	0,21	2,70	0,28	0,18	0,35
Bounded mean (15%)	1,50	6,53	12,33	0,18	2,58	0,15	0,09	0,28
Q3	2,00	9,75	15,75	0,50	3,75	0,00	0,12	1,00
Q2 Median	1,00	4,50	8,00	0,00	2,00	0,00	0,00	0,00
Q1	0,00	2,00	4,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	3,59	10,35	20,42	0,33	1,84	0,72	0,45	0,60
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	18,00	62,00	106,00	0,91	6,00	3,00	2,00	3,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Junit 3.2	2,72	7,94	14,75	0,25	2,56	0,19	0,13	0,34
Bounded mean (15%)	1,68	5,96	11,29	0,22	2,43	0,04	0,09	0,25
Q3	3,00	11,00	18,50	0,50	3,50	0,00	0,19	1,00
Q2 Median	1,00	3,50	5,50	0,00	2,00	0,00	0,00	0,00
Q1	0,00	2,00	2,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	4,85	11,65	21,05	0,34	1,93	0,64	0,24	0,65
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	20,00	60,00	103,00	0,92	6,00	3,00	0,75	3,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Junit 2.1	2,16	8,11	14,05	0,22	2,53	0,32	0,31	0,58
Bounded mean (15%)	1,35	7,18	12,41	0,19	2,41	0,18	0,17	0,47
Q3	2,00	8,50	18,00	0,50	3,00	0,00	0,26	1,00
Q2 Median	1,00	4,00	6,00	0,00	2,00	0,00	0,00	0,00
Q1	0,00	4,00	4,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	4,06	8,52	15,30	0,31	1,61	0,82	0,70	0,84
Minimum	0,00	1,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	18,00	31,00	55,00	0,89	6,00	3,00	3,00	3,00

viación estándar, cuartil inferior (Q1), mediana (Q2), cuartil superior (Q3), mínimo y máximo. En las Fig.3.4, 3.5, 3.6 se observa que los valores umbrales de cada producto mantienen los valores umbrales en su evolución. Esta observación sugiere que productos estables, incluso duplicando el número de clases, mantienen sus valores umbrales.

Las conclusiones extraídas de este estudio empírico son:

- Los valores umbrales deberían ser relativos al producto concreto
- Los valores umbrales se mantienen a partir de versiones estables
- Parece que el tipo de software (framework de dominio, framework de aplicación o biblioteca) no determina cómo fijar los valores umbrales

3.2. Propuesta de un proceso de medición

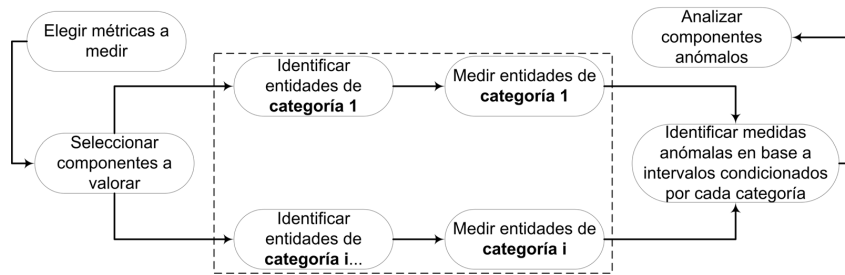
El éxito del proceso de medición expuesto en la Figura 3.1 es que posibilita identificar componentes anómalos. En este sentido, en la Sec. 3.1 se ha justificado la utilización de valores umbrales relativos al producto concreto, pero se han dejado abiertas dos cuestiones que pueden influir en el proceso:

- Existe una gran dispersión entre las medidas, por tanto los valores umbrales pueden ser poco precisos
- En desarrollos iniciales de productos, se necesita estimar valores umbrales específicos

Las entidades a medir pueden ser muy diferentes, de acuerdo a la naturaleza del problema que resuelven. Nos referiremos a éstas como entidades de diferentes categorías. Como ya se ha mencionado una aportación de esta tesis es considerar que esta información puede mejorar el proceso de medición que permite identificar entidades anómalas. Por ello proponemos, un nuevo proceso de medición (ver Fig. 3.7) que validaremos de forma empírica en la Sec. 3.3.

La modificación que proponemos en el proceso de medición de un producto software, tiene como objetivo, mejorar la identificación de componentes anómalos. Proponemos añadir al proceso de medición la categoría de la entidad a medir, y como consecuencia, habrá que utilizar diferentes valores umbrales dependiendo de la categoría de la entidad. Cuando se tienen en cuenta las categorías de la entidad, se produce un desglose de tareas en el proceso de medición, conducentes a obtener diferentes valores umbrales para cada categoría. En la Figura 3.7 se muestra la adaptación del proceso clásico de medición cuando se incorporan las nuevas tareas, que aparecen enmarcadas en un rectángulo.

Un tipo de categoría que puede ayudar a definir la naturaleza del problema que resuelve una entidad es el que puede indicarse con ciertos estereotipos estándar UML. En el diseño de un sistema software orientado a objetos la información de estereotipos UML muchas veces no está disponible explícitamente en los modelos o en el código, aunque los diseñadores y programadores la hayan tenido en cuenta implícitamente en sus soluciones. Un ejemplo de esto son los estereotipos sobre las clases de análisis [AN05, JBR00], entidad (*entity*), controlador (*control*) y límite (*boundary*). Además, el criterio de clasificación límite se desglosa a su vez en: interfaz de usuario, interfaz de sistema e interfaz de dispositivo. Otros estereotipos interesantes en los clasificadores, son los obtenidos como resultados de algunas tareas del proceso de desarrollo, como los relacionados con excepciones, pruebas y utilidades.

Figura 3.7. Proceso de medición propuesto

Si se considera un contexto del proceso de medición donde el producto a medir son códigos fuente, obtener los estereotipos manualmente es una tarea ingente y exigiría gran cantidad de recursos, por la cantidad de entidades de código en un sistema real. Por esto son necesarios algoritmos y herramientas que asistan a la clasificación automática de entidades de código. Con este cometido en [LMC10] se presentó un plugin de Eclipse que clasifica las entidades a partir de un algoritmo basado en convención de nombres de las entidades. Posteriormente en la Sec.3.4.3 se describe el algoritmo propuesto y en el Cap. 5 se hace una descripción más detallada de la herramienta.

En resumen, la modificación del proceso de medición propuesto se basa en considerar la naturaleza de la entidad de código, que en este estudio será uno de los siguientes estereotipos UML: *e1 exception*, *e2 interface*, *e3 entity*, *e4 control*, *e5 test*, *e6 utility*.

3.3. Estudio empírico para validar el proceso de medición

Para validar los cambios propuestos en el proceso de medición, planteamos la hipótesis de que los valores umbrales de las mediciones de entidades de código, están condicionados por la naturaleza del problema que resuelve cada entidad de código, entendiendo por naturaleza aquella que se expresa con estereotipos estándar de clasificadores UML. Para corroborar esta hipótesis, se van a realizar los siguientes estudios observacionales, investigando en contextos reales de desarrollo del software.

En las Sec. 3.4, 3.5, 3.6 se van a presentar dos casos de estudio cada uno con dos réplicas. El primero tiene como objetivo:

Estudiar si las métricas de código se comportan igual en las entidades, independientemente de su naturaleza.

En este caso de estudio hay una amenaza como consecuencia de la subjetividad del método de clasificación automática de entidades. Por esa razón planteamos el segundo caso de estudio, cuyo objetivo es:

Comprobar si el método de clasificación realizada por humanos está relacionado con una clasificación automática.

El desarrollo de los casos de estudio ha seguido las guías para conducir la experimentación en ingeniería del software expuestas en [RH09] y el proceso experimental propuesto en [WRH⁺00], para el que se especifican las siguientes fases: definición, planificación, operación, análisis, validez y presentación.

En la Tabla 3.1 se caracterizan de manera resumida los casos de estudio realizados, considerando el número de sujetos y objetos que intervienen, el objetivo y la sección de este documento donde se presentan. Como número de objetos se ha considerado el número de proyectos tratados, cada uno de los cuales tendrá un número de entidades de código.

Tabla 3.1. Caracterización de los casos de estudio

Caso de estudio	#Sujetos	#Objetos	Objetivo	Sección
Plugings de Eclipse	3	10	Estudiar si las métricas de entidades de código se comportan igual, independientemente de la naturaleza de la entidad	3.4
Réplica Trabajos Fin de Carrera	1	30		3.5
Original DMSII	8	2	Estudiar si el método de clasificación automático de entidades de código no es concordante con una clasificación realizada por personas	3.6
Réplica GI-RO	5	2		3.6

3.4. Caso de estudio 1: Medidas en Plugings de Eclipse

En esta sección se presenta un caso de estudio con un doble objetivo:

- Definir una clasificación de entidades de código según la naturaleza de diseño que se pueda automatizar
- Estudiar la relación entre la clasificación y las medidas de las entidades

Los objetos para obtener la clasificación son proyectos de código abierto y los sujetos son un experto y dos estudiantes. Los resultados obtenidos han sido la clasificación de las entidades de código examinadas, realizada por humanos y la constatación empírica de la existencia de relación entre dicha clasificación y las medidas obtenidas en las entidades de código del estudio.

3.4.1. Definición del caso de estudio: objetivos, hipótesis y variables

El estudio tiene dos objetivos:

Objetivo 1

- Analizar entidades de código fuente
- Con el propósito de obtener una clasificación
- Con respecto a la naturaleza de entidades basada en estereotipos UML
- Desde el punto de vista de los investigadores
- En el contexto de un conjunto de aplicaciones de código abierto, en concreto, plugins de Eclipse, en la Universidad de Burgos.

Objetivo 2

- Analizar entidades de código fuente
- Con el propósito de estudiar la relación de la naturaleza de dichas entidades
- Con respecto a las métricas de tamaño, documentación, acoplamiento, cohesión, herencia y complejidad
- Desde el punto de vista de los investigadores
- En el contexto de un conjunto de aplicaciones de código abierto, en concreto, plugins de Eclipse, en la Universidad de Burgos.

Los objetivos de este estudio dan lugar a la siguiente hipótesis nula:

*Las métricas de entidades de código se comportan igual,
independientemente de la naturaleza de la entidad*

Las variables dependientes son las medidas de entidades de código descritas en la Tabla 3.2 (M_i). La variable independiente es la clasificación de dichas entidades, basada en la siguiente escala nominal: *e1 exception, e2 interface, e3 entity, e4 control, e5 test, e6 utility*.

3.4.2. Planificación

Esta sección presenta la información relativa a los sujetos y objetos del estudio y cuestiones relacionadas con la instrumentación para llevar a cabo el estudio.

3.4.2.1. Selección del contexto y sujetos

El caso de estudio se llevó a cabo como trabajo en una asignatura de 5^o de Ingeniería Informática. En el trabajo participaron dos estudiantes para la recogida de mediciones y un tutor especializado en el área de mantenimiento del software.

En el contexto global de la asignatura, los tutores proponen varios trabajos distintos para grupos de dos estudiantes, uno de ellos fue este caso de estudio. Posteriormente los estudiantes de la asignatura eligen uno o varios. La asignación final del trabajo al grupo único de alumnos sigue la siguiente regla: en caso de existir más de un grupo interesado en el mismo proyecto, el profesor elige el de mejor expediente académico.

3.4.2.2. Objetos del estudio

Para seleccionar los proyectos que son los objetos de este estudio, hemos considerado únicamente plugins de la herramienta Eclipse, obtenidos a través del repositorio de software de código abierto *SourceForge* (<http://sourceforge.net/>). La selección de proyectos se hizo siguiendo los criterios siguientes:

- Porcentaje de actividad, medido a través de la información de actividad de modificaciones continuadas y la actividad reciente. El criterio establecido fue porcentaje de actividad ≥ 85
- Popularidad, medida a través del número de descargas por parte de los usuarios. El criterio establecido fue N° de Descargas ≥ 7000 .
- Estado del desarrollo de la aplicación, medido utilizando la siguiente escala ordinal: 1 Planificación, 2 PreBeta, 3 Alpha, 4 Beta, 5 Producción/Estable, 6 Madurez, 7 Inactive. El criterio establecido fue estado ≥ 3 .
- Lenguaje de programación utilizado en su implementación, aplicando el criterio de que sea el mismo lenguaje para todos, Java en este caso.

3.4.2.3. Diseño del caso de estudio

Todos los sujetos implicados trabajaron con todos los objetos utilizados, con el fin de obtener la clasificación de los mismos, para ello utilizaron una convención de nombres (Tabla 3.4). De esta forma se cubre el primer objetivo del estudio. Los alumnos proponen cadenas de texto que ayudan a identificar entidades de cada clase y el tutor valida dichas cadenas con inspecciones sobre el código de entidades, que contenían las cadenas en sus nombres. Este proceso de validación es iterativo, hasta conseguir clasificar más del 50% de las entidades con el conjunto de nombres proporcionado. Una vez obtenidas las cadenas de cada categoría considerada, se procedió a automatizar el proceso de clasificación.

3.4.2.4. Instrumentación

Este experimento requiere dos tipos de herramientas, unas para calcular métricas de código y otras que asistan en el proceso de categorización de las entidades de código.

En el contexto de medición que se plantea en este trabajo, se necesita una herramienta que calcule un amplio conjunto de métricas orientadas a objetos, y permita la exportación de resultados para su análisis posterior. Bajo estas premisas, la herramienta

seleccionada para obtener las medidas ha sido RefactorIt [Aqr02]. En la Tabla 3.2 se presentan las métricas proporcionadas por la herramienta y los valores umbrales que recomienda RefactorIt para algunas de ellas (columnas MinValor y MaxValor). Además, en la columna llamada ámbito se presenta a qué tipo de entidad de código está asociada la métrica: paquete (P), clase (C), método (M) o todas las categorías anteriores juntas (T).

Tabla 3.2. Conjunto de métricas definido en RefactorIt

Descripción	Identificador	MinValor	MaxValor	Ámbito	Característica
Comment Lines of Code	CLOC			T	TAM
Cyclomatic Complexity	V(G)	1	10	M	COM
Density of Comments	DC	0.2	0.4	T	DOC
Executable Statements	EXEC	0	20	T	TAM
Non-Comment Lines of Code	NLOC			T	TAM
Number of Parameters	NP	0	4	M	TAM
Total Lines of Code	LOC	5	1000	T	TAM
Abstractness	A	0.0	0.5	P	ABS
Afferent Coupling	Ca	0	500	P	ACO
Depth in Tree	DIT	0	5	C	HER
Efferent Coupling	Ce	0	20	P	ACO
Instability	I	0.7	1.0	P	ACO
Number of Abstract Types	NOTa	0	20	P	ABS
Number of Children	NOC	0	10	C	HER
Number of Concrete Types	NOTc	0	80	P	ABS
Number of Exported Types	NOTe	3	50	P	ACO
Number of Fields	NOF	0	1	C	TAM
Number of Types	NOT	0	80	P	TAM
Response for Class	RFC	0	50	C	COM
Weighted Methods per Class	WMC	1	50	C	COM
Number of Attributes	NOA	0	5	C	TAM
Cyclic Dependencies	CYC	0	1	P	PDA
Dependency Inversion Principle	DIP	0.3	1.0	C	PDA
Direct Cyclic Dependencies	DCYC	0	1	P	PDA
Distance from the Main Sequence	D	0.0	0.1	P	PDA
Encapsulation Principle	EP	0	0.6	P	PDA
Lack of Cohesion of Methods	LCOM	0.0	0.2	C	COH
Limited Size Principle	LSP	0	10	P	PDA
Modularization Quality	MQ	0	1000	P	PDA
Number of Tramps	NT	0	1	M	O

Independientemente de las convenciones particulares sobre las métricas de código, cada vez que se analiza o se habla del código de un sistema software, se quiere obtener información del tamaño y complejidad del mismo. Algunos trabajos expresan el tamaño de un sistema en términos de líneas de código, número de clases e incluso cantidad

de megabytes del código fuente. Desde nuestra perspectiva, la caracterización de las entidades por medio de métricas es de interés en cuanto que éstas reflejen la bondad de ciertos aspectos del diseño como son: tamaño (TAM), documentación (DOC), acoplamiento (ACO), herencia (HER), complejidad estructural (COM), abstracción (ABS), cohesión (COH) o principios de diseño (PDA). Se ha incorporado una columna en la Tabla 3.2, llamada *característica*, que recoge esta clasificación subjetiva. En cualquier caso, estas métricas son consideradas de interés por su relación con diferentes aspectos de la calidad de los productos software [BEM95, Dro96, Moo05, Man09].

3.4.3. Operación

La instrumentación ha sido realizada en ordenador personal en el mes de febrero de 2009. Los alumnos han sido formados en el uso de la herramienta RefactorIt por el mismo profesor que les guiará en el caso de estudio. Además seleccionaron los proyectos a medir bajo los criterios propuestos por el tutor. En cada fila de la Tabla 3.3 se presentan las características de los proyectos elegidos en este caso de estudio. Por cada uno se presenta información relativa a su tamaño, expresada como número de entidades de cada categoría, e información externa obtenida por el repositorio software de código abierto. En la columna *sin clasificar* se indica el número de entidades que no han podido ser clasificadas en ninguna de las categorías. La última fila y la última columna muestran el total de líneas de código, por cada tipo de entidad y cada proyecto, respectivamente.

Tabla 3.3. Información de los proyectos elegidos

	Número de entidades							Información Sourceforge			Tamaño
	e ₁	e ₂	e ₃	e ₄	e ₅	e ₆	Sin clasificar	%Actividad	Nº Descargas	Estado	LOC
Plugins Eclipse											
esFtp	1	90	38	68	8	86	52	94,19	45878	4	5156
AVR	12	376	125	306	362	789	310	98,68	1464780	5,6	55004
Jedit	25	1671	1657	1548	12	1719	681	99,62	5371954	5,6	156656
EclEmma	1	257	288	78	186	35	150	99,93	1488061	5	13294
AzSMRC	45	511	655	272	9	759	758	99,00	59327	4,5	67760
EclipseME	39	575	951	535	281	1215	446	97,63	731177	5	80012
ELBE	26	1788	1561	1216	138	380	288	85,95	30172	7	76188
OpenReports	17	0	568	1074	2	159	383	96,80	235708	4,5	27338
EclipseCorba	7	145	148	232	143	205	334	94,86	23062	3	25051
LabelDecorator	7	0	116	34	52	177	74	85,00	7004	5	5256
LOC Totales	909	111818	85911	83876	14238	14797	200166				

Para obtener la clasificación de entidades, *objetivo 1* de este estudio, inicialmente se ha partido de un algoritmo de clasificación que se basa en criterios de convención de nombres de las entidades. Todas las entidades de código tratadas tienen un nombre *simple* y un nombre cualificado. El nombre cualificado es el nombre *simple* precedido de los nombres *simples* de todos los espacios de nombre donde está contenido el elemento.

En la Tabla 3.4 se recoge la convención de nombres utilizada, de manera que una entidad se clasifica en una de esas categorías, si contiene en su nombre cualificado alguna de las cadenas referenciadas en la tabla. En el caso de conflicto, porque el nombre contenga cadenas que correspondan a más de una categoría, prevalece el criterio del nombre simple de la entidad. Aunque las cadenas de la tabla de convención de nombres se presentan en inglés, se aplica una traducción de los mismos términos en distintos idiomas. En Alg. 1 se presenta una descripción en pseudocódigo del algoritmo. En [LMC10] se presentó un plugin de Eclipse que ejecuta el algoritmo utilizado en este estudio.

Algoritmo 1 Clasificación de entidades de código en categorías basado en el nombre de la entidad

Entrada: Tabla de Convención de Nombres (*tcn*) que asocia cadenas de caracteres a categorías (ver Tabla 3.4)

Entrada: Entidad de código que se quiere clasificar

Salida: Una categoría de la Tabla de Convención de Nombres (*tcn*): *e1 excepción, e2 interfaz, e3 entidad, e4 control, e5 test, e6 utilidad, e7 desconocida*

```

1: Obtener nombre simple y nombre cualificado de la entidad
2: Buscar si nombre cualificado contiene cadenas en tcn
3: if nombre cualificado no contiene ninguna cadena then
4:   return categoría e7 desconocida
5: end if
6: if nombre cualificado contiene cadenas de una sólo categoría then
7:   return la categoría asociada
8: end if
9: if nombre cualificado contiene cadenas de varias categorías then
10:  Buscar si el nombre simple contiene cadenas en tcn
11:  if nombre simple contiene cadenas de una sólo categoría then
12:    return categoría asociada
13:  end if
14: else
15:  entidadTratada = entidad contenedora de la entidad
16:  while exista entidadTratada do
17:    Obtener nombre simple de la entidadTratada
18:    Buscar si nombre simple contiene cadenas en tcn
19:    if nombre simple contiene cadenas de una sola categoría then
20:      return la categoría asociada
21:    else
22:      entidadTratada = entidad contenedora de la entidadTratada
23:    end if
24:  end while
25: end if
26: return categoria e7 desconocida

```

El tutor del experimento validó la Tabla 3.4, tal y como se describe en la Sec. 3.4.2.3. Los alumnos realizaron después las tareas siguientes:

Tabla 3.4. Convención de nombres en inglés para clasificar entidades

Categorías	e_1	e_2	e_3	e_4	e_5	e_6
Convención de nombres	exception	interface gui forms ui report swing visual view awt	core model entity	control facade manager handler action callback maker provider	test debug dummy	util properties log preference template options

1. Realizaron la medición de las entidades con la herramienta RefactorIt
2. Exportaron los datos de las entidades y sus medidas a hojas de cálculo
3. Aplicaron las técnicas de agrupamiento y convención de nombres para obtener las clasificaciones de las entidades a partir de los datos de la Tabla 3.4

La gran cantidad de entidades medidas y clasificadas en la recolección de datos de este caso de estudio (27.256) impide incluirlas en este documento. El conjunto de datos completo generado para su posterior análisis puede ser obtenido en [Lóp12a].

3.4.4. Análisis

Algunas de las entidades seleccionadas (39,2%) no se pudieron clasificar por diferentes razones: porque la entidad pertenecía a varias categorías o porque la técnica utilizada para clasificar no es exhaustiva, y hay entidades que no se pueden clasificar. El porcentaje de entidades clasificadas en alguna de las categorías ha sido del 60,8%, tomando como unidad la línea de código, y son estas entidades las consideradas en el estudio que sigue.

La hipótesis que vamos a estudiar es la siguiente:

H_0 : La métrica M_i se comporta igual en las entidades de código, independientemente de su naturaleza.

H_1 : La métrica M_i se comporta de manera diferente, dependiendo de la naturaleza de las entidades de código.

Donde M_i es una de las métricas de la Tabla 3.2, y la naturaleza de la entidad las seis categorías de la Tabla 3.4, que definen los niveles del tratamiento. Un test de ANOVA (ANALYSIS OF VARIANCE) es el adecuado para contrastar este tipo de hipótesis, siempre y cuando se cumplan ciertas condiciones [WBM91]: Normalidad de M_i y homocedasticidad. Si alguna de las condiciones no se cumple, usaremos el test similar, no paramétrico, de Kruskal-Wallis [SC88], en adelante K-W. Para estudiar la Normalidad usaremos el test de Shapiro Wilk o el de Kolmogorov Smirnov si la muestra es mayor de 50 elementos. Con respecto a la homocedasticidad, emplearemos el test de homogeneidad de varianzas de Barlett.

Si el resultado del test, paramétrico o no, es significativo se rechazará la hipótesis nula, aceptando que la naturaleza de la entidad condiciona los valores de la métrica, y por ello sus valores umbrales. En este caso tendrá sentido proporcionar valores umbrales de la métrica para cada categoría considerada, valores umbrales que estimaremos con los percentiles 25 y 75 [FP97] para cada uno de los estereotipos considerados. Estos umbrales son de propósito general, y se podían haber obtenido a partir de límites de confianza, usando la desigualdad de Tchebychev o la distribución Normal, dependiendo del tipo de distribución de M_i . Los contrastes de hipótesis mencionados se han realizado utilizando el software R [OS09].

3.4.4.1. Resultados observados

Puesto que los contrastes de normalidad dieron significativos, no podemos aceptar que la distribución sea Normal. Es por ello que utilizaremos el contraste de K-W. Para cada una de las métricas de código (M_i) consideradas, se enuncia la hipótesis siguiente:

$$H_0 : \mu_{e1}^{M_i} = \mu_{e2}^{M_i} = \mu_{e3}^{M_i} = \mu_{e4}^{M_i} = \mu_{e5}^{M_i} = \mu_{e6}^{M_i}$$

donde $\mu_{e_j}^{M_i}$ representa la media de los valores de la métrica M_i correspondientes a las entidades con estereotipo e_j .

En la Tabla 3.5 se presenta un resumen de los resultados obtenidos al aplicar el test no paramétrico K-W para cada una de las métricas consideradas, cuando las entidades de código están clasificadas por estereotipos.

Para facilitar el análisis, la Tabla 3.5 representa dos criterios de clasificación respecto a las métricas (ver Sec. 3.4.2.4): por un lado el ámbito de aplicación de la entidad que mide (paquete, clase, método) y por otro lado, la relación subjetiva con ciertas características del sistema relacionadas con su calidad (TAM, PDA, O, HER, DOC, COM, COH, ACO, ABS). Las filas de la tabla están ordenadas por el segundo criterio. Las tres últimas columnas recogen las tres categorías del criterio de ámbito. Respecto al significado del contenido literal de las celdas “- - -” que aparece en la Tabla 3.5, indica que la métrica no es aplicable en ese ámbito.

En la Tabla 3.5 se muestran los p-valores del test de K-W. Puede verse que 34 de los 39 resultados han sido significativos al nivel 0,05. Por ello, en la mayoría de las métricas estudiadas podemos aceptar que los valores umbrales debemos obtenerlos teniendo en cuenta la naturaleza de la entidad medida.

Si se realiza este análisis utilizando el criterio de ámbito, se observa que en el ámbito de paquete 3 métricas son no significativas (no se rechaza H_0) frente a 16 significativas (se acepta H_1), en el ámbito de clase y método la hipótesis H_1 se afianza, 18 cumplen H_1 frente a 2 H_0 . Desde el criterio de característica se afianza H_1 en las métricas de complejidad (COM), documentación (DOC) y abstracción (ABS) (cero no significativos y tres significativos), respecto al acoplamiento (ACO) (uno no significativo y tres significativos), la aceptación de H_1 es más moderada. Respecto a la categoría de principios de diseño (PDA) la aceptación de H_1 es más débil (tres no significativos y cuatro significativos). En la Tabla 3.6 se muestra un resumen de los resultados significativos.

Tabla 3.5. p-valores observados en el test de K-W, subrayados los no significativos al nivel 0.05

Identificador	Característica	Resultados test K-W		
		Paquete	Clase	Método
NOT	TAM	0.02238	---	---
NP	TAM	---	---	< 2.2e-16
NOA	TAM	---	< 2.2e-16	---
NLOC	TAM	0.01340	< 2.2e-16	< 2.2e-16
EXEC	TAM	0.005242	< 2.2e-16	< 2.2e-16
CLOC	TAM	0.02767	< 2.2e-16	0.0129
MQ	PDA	<u>0.1427</u>	---	---
LSP	PDA	<u>0.0775</u>	---	---
EP	PDA	0.006352	---	---
D	PDA	0.0004983	---	---
DCYC	PDA	0.03573	---	---
DIP	PDA	---	<u>0.2769</u>	---
CYC	PDA	0.0002174	---	---
NT	O	---	---	< 2.2e-16
NOC	HER	---	<u>0.2813</u>	---
DIT	HER	---	< 2.2e-16	---
DC	DOC	0.01089	< 2.2e-16	0.0367
WMC	COM	---	< 2.2e-16	---
RFC	COM	---	2.68E-09	---
V(G)	COM	---	---	< 2.2e-16
LCOM	COH	---	< 2.2e-16	---
NOTe	ACO	<u>0.1909</u>	---	---
I	ACO	0.0001181	---	---
Ce	ACO	0.007674	---	---
Ca	ACO	0.003002	---	---
NOTc	ABS	0.001988	---	---
NOTa	ABS	0.03124	---	---
A	ABS	0.0005088	---	---

3.4.4.2. Valores umbrales dependientes de la naturaleza de la entidad

Como resultado final del caso de estudio, las Tablas 3.7, 3.8 y 3.9 recogen los valores umbrales para aquellas métricas cuyos resultados son favorables a la hipótesis H_1 . Los valores umbrales propuestos se han obtenido seleccionando percentiles 25 (Q1) y 75 (Q3), de cada métrica [FP97]. Basándonos en estos valores umbrales propuestos, se expone un escenario posible para utilizar esta información con el fin de identificar medidas anómalas. En la Tabla 3.8 se ha resaltado la columna de la métrica de com-

Tabla 3.6. Resumen de resultados significativos al nivel 0.05

	# significativos	# totales	%
Global			
	34	39	87 %
Ámbito			
Paquete	16	19	84 %
Clase	9	11	82 %
Método	7	7	100 %
Características			
Complejidad	3	3	100 %
Documentación	3	3	100 %
Abstracción	3	3	100 %
Tamaño	12	12	100 %
Acoplamiento	3	4	75 %
Herencia	1	2	50 %
Principios de diseño	3	7	43 %

plejidad estructural *WMC* (*Weighted Methods per Class*) [CK94], donde se indican los valores umbrales para cada uno de los estereotipos considerados $e1$ [1.75 - 4], $e2$ [4 - 16], $e3$ [5 - 25], $e4$ [3 - 16.25], $e5$ [2 - 8.25], $e6$ [4 - 22], mientras que en la Tabla 3.2 aparece [1, 50] como valores umbrales recomendados por la herramienta RefactorIt, para esa métrica. A partir de esta información podemos destacar dos cosas: por un lado, el intervalo propuesto por la herramienta es menos preciso que los proporcionados por nosotros, de hecho los incluye. Por otro lado, la precisión (amplitud) de los valores umbrales depende de las categorías consideradas; de hecho en las clases excepción ($e1$) y test ($e5$) la precisión es mejor que en las restantes. Análogamente podemos realizar este análisis sobre el resto de las métricas consideradas, llegando a conclusiones similares a las anteriores.

3.4.5. Validación

Los resultados observados son aplicables a una población de entidades similares a las seleccionadas en el estudio. Para generalizar estos resultados puede ser interesante replicar este estudio con un conjunto de métricas distinto.

Hemos detectado las siguientes amenazas a la validez de los resultados. Por una parte la subjetividad de la clasificación, que es difícilmente evitable. Por otro lado, la no supervisión de los sujetos. Ambas amenazas pueden resolverse, perfeccionando la clasificación y supervisando el estudio.

Tabla 3.7. Métricas de paquetes: Valores umbrales recomendados según la naturaleza del problema

		EXEC	Ca	Ce	I	A	D	NOTc	CYC	EP
Exception e ₁	Q1	---	---	---	---	---	---	---	---	---
	Q3	---	---	---	---	---	---	---	---	---
Interfaz e ₂	Q1	15.00	0.0	5.00	0.60	0.00	0.00	4	0	0.071
	Q3	134.00	7.0	18.00	1.000	0.07	0.36	18	3.0	0.80
Entity e ₃	Q1	11.75	2.0	3.00	0.25	0.00	0.10	2	0	0.52
	Q3	194.50	35.5	14.00	0.68	0.26	0.50	12	4.5	1
Control e ₄	Q1	7.50	0.0	2.00	0.31	0.00	0.00	1	0	0.00
	Q3	93.00	8.5	11.00	1.00	0.29	0.25	11	1.5	1
Test e ₅	Q1	2.00	0.0	2.00	0.93	0.000	0.00	1	0	0.00
	Q3	28.00	1.0	7.75	1.0	0.39	0.34	7	0	0.19
Utility e ₆	Q1	12.75	0.0	2.00	0.45	0.00	0.00	2	0	0.000
	Q3	127.25	6.0	14.00	1.00	0.17	0.40	13	1.0	0.81

Tabla 3.8. Métricas de clases: Valores umbrales recomendados según la naturaleza del problema

		DC	LOC	NCLOC	EXEC	CLOC	WMC	DIT	RFC	LCOM	NOA
Exception e ₁	Q1	0.00	10.25	6.25	0	0	1.75	3	1.75	0	0
	Q3	0.14	29.25	15.00	1.75	3	4.00	4	3.25	0.00	2
Interfaz e ₂	Q1	0.00	35.00	24.00	2	0	4.00	1	2.00	0	1
	Q3	0.20	179.25	133.25	14.00	25	16.00	2	19.00	0.95	7
Entity e ₃	Q1	0.00	40.00	24.75	2	0.75	5.00	1	2.00	0	1
	Q3	0.27	169.50	119.25	17.00	32	25.00	2	21.00	0.96	6
Control e ₄	Q1	0.00	18.00	14.00	1	0	3.00	1	2.00	0	0
	Q3	0.21	115.25	79.00	12.00	13	16.25	2	17.00	0.75	3
Test e ₅	Q1	0.00	19.00	12.25	0	0	2.00	1	2.00	0	0
	Q3	0.28	105.00	71.25	8.00	19	8.25	2	9.00	0.81	2
Utility e ₆	Q1	0.02	33.00	22.00	2	1	4.00	1	3.00	0	1
	Q3	0.31	202.75	130.75	23.00	41	22.00	2	23.00	0.92	6

3.5. Réplica del caso de estudio 1: Mediciones en trabajos fin de estudios

En esta sección se presenta una réplica del caso de estudio de la Sec. 3.4. Esta réplica sirve para confirmar, o no, los resultados del caso de estudio de la Sec. 3.4, usando nuevos objetos de estudio (proyectos fin de estudios) y métricas diferentes. Se trabaja en un contexto próximo al inspector, en concreto con códigos de trabajos fin de carrera de 5º de Ingeniería Informática de la Universidad de Burgos. La clasificación se realiza por un inspector que ha participado en la formación de los desarrolladores y que ha tenido que evaluar sus trabajos en un tribunal.

Tabla 3.9. Métricas de métodos: Valores umbrales recomendados según la naturaleza del problema

		LOC	NCLOC	EXEC	NP	V(G)	NT
Exception e ₁	Q1	1	1	0	0	1	0
	Q3	8	7.75	1	2	2	0.75
Interfaz e ₂	Q1	1	1	0	0	1	0
	Q3	14	12	2	1	2	0
Entity e ₃	Q1	1	1	0	0	1	0
	Q3	9	8	2	1	2	0
Control e ₄	Q1	1	1	0	0	1	0
	Q3	11	9	2	1	3	0
Test e ₅	Q1	1	1	0	0	1	0
	Q3	10	8	1	1	1	0
Utility e ₆	Q1	1	1	0	0	1	0
	Q3	14	11	3	1	3	0

3.5.1. Definición de la réplica del caso de estudio 1

Exceptuando el contexto de los objetos, donde se cambian los plugins de Eclipse por Trabajos fin de estudio, los objetivos son idénticos a los dos del caso de estudio y especificados en la Sec. 3.4.1

Los objetivos de este estudio dan lugar a la siguiente hipótesis nula:

*Las métricas de entidades de código se comportan igual,
independientemente de la naturaleza de la entidad*

Las variables dependientes son las medidas de entidades de código de la Tabla 3.11 (M_i). La variable independiente es la clasificación de dichas entidades, basada en la siguiente escala nominal: *e1 exception, e2 interface, e3 entity, e4 control, e5 test*.

3.5.2. Planificación

Esta sección presenta la información relativa a los sujetos y objetos del estudio y cuestiones relacionadas con la instrumentación para llevar a cabo el estudio.

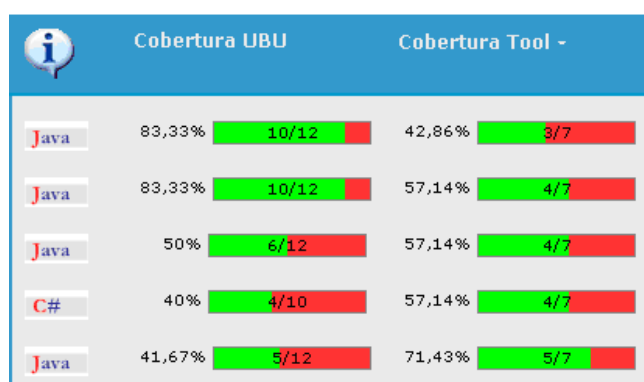
3.5.2.1. Selección del contexto

En el 5º curso de Ingeniería Informática en la Universidad de Burgos (UBU) existe una asignatura de Sistemas Informáticos conocida comúnmente como proyecto fin de carrera. En ella, los alumnos deben presentar un proyecto obtenido fruto de un proceso de desarrollo. Como responsables de la calificación de los mismos, se establecieron unos criterios de evaluación, algunos de los cuales se refieren a la valoración de la calidad del código. Para ello, se utilizan métricas de código recogidas automáticamente a través de herramientas. Su interpretación se hace a través de valores umbrales generales y

otros relativos a la Universidad de Burgos. Como criterio de conformidad se utiliza un indicador de cobertura de métricas, que se obtiene contando el número de los valores de las métricas de un proyecto que están dentro de los valores umbrales recomendados, dividido por el número total de métricas consideradas.

En la Fig. 3.8 se muestra la evaluación de cinco de proyectos (filas de la tabla) utilizando dos indicadores de cobertura, uno relativo a la propia organización UBU y otro general basado en los valores umbrales recomendados por la herramienta. Además, se muestran los valores ordenados respecto al indicador general (columna Cobertura Tool). Un histórico de los datos de métricas de distintos trabajos puede obtenerse en [Lóp12a].

Figura 3.8. Evaluación de proyectos



En este proceso de evaluación, aparecieron algunos casos en los cuales se obtuvieron resultados contradictorios entre las coberturas de las métricas y los criterios de evaluación que utilizaban los miembros del tribunal evaluador. La experiencia adquirida es, que la interpretación de un valor de una métrica utilizando un sólo intervalo general, o relativo a la propia organización, no es suficientemente fiable. La interpretación puede mejorar si se incorpora al proceso de medición información de la naturaleza del problema que resuelve el proyecto medido.

Hasta donde llega nuestro conocimiento, no existen clasificaciones de proyectos basadas en la naturaleza del problema. Pero cuando se realiza una rápida inspección de código, tanto los desarrolladores como los evaluadores de una organización sí que conocen la naturaleza del problema que resuelven las entidades de código del proyecto. Así, a través de las inspecciones realizadas basadas en la estructura del código, se pueden distinguir las capas y subsistemas del proyecto: interfaz gráfico, modelo, controladores, pruebas. Además dentro de las capas y subsistemas, existen unas clases especiales, llamadas excepción, para el tratamiento de errores.

3.5.2.2. Selección de objetos

El proceso de muestreo seguido ha sido aleatorio respecto a los proyectos presentados en la asignatura. Se seleccionaron 30 proyectos presentados en la asignatura de Sistemas

Informáticos de 5º de Ingeniería Informática de la Universidad de Burgos desde el curso 2003-04, además de 2 proyectos desarrollados por profesores de la misma Universidad y 2 de carácter industrial, Java Development Kit 1.5.0, JUnit 4.0 [Lóp12a]. Estos cuatro últimos sirven para comparar los trabajos de los alumnos, y como criterio de validación externo del caso de estudio.

3.5.2.3. Instrumentación

Este experimento necesita herramientas, para calcular métricas de código que se adapten al contexto definido. Para ayudar a la elección de una, en la Tabla 3.10 se muestra el resultado de la evaluación de un conjunto de herramientas respecto a las siguientes características:

- C1. Lenguaje de programación sobre el que trabajan
- C2. Entrada: ficheros binarios o fuentes (binarios/fuentes/ambos)
- C3. Número de métricas que calcula
- C4. Formato de exportación de resultados (html/txt/xml/xls)
- C5. Indicadores gráficos o técnicas de agrupación y filtrado para analizar los resultados (Si/No)

Tabla 3.10. Herramientas de medida de código

Herramientas	C1	C2	C3	C4	C5
Dependency Finder	java	binarios	33	html,txt,xml	No
RefactorIt	java	fuentes	25	html,txt,xml	Si
JDepend	java	binarios	9	html,txt,xml	No
Eclipse Metrics - v1.3.6	java	fuentes	25	xml	No
NDepend	.NET	ambos	66	html, txt, xml, xls	Si
SourceMonitor	java, C#, C++, VB	fuentes	14	txt, xml,	Si

Aunque las definiciones de muchas de las métricas de código se mantienen independientes del lenguaje de programación, en la práctica muchas herramientas o componentes sólo trabajan sobre un único lenguaje de programación (ver columna C1 de la Tabla 3.10). En el contexto de medición que se plantea en este caso de estudio, se debe permitir evaluar proyectos de diferentes lenguajes de programación, y la exportación de resultados en formatos de texto para el posterior análisis de los mismos. Siguiendo esos criterios, la herramienta seleccionada es SourceMonitor [CS07].

En la Tabla 3.11 se presentan las métricas de subsistema proporcionadas por la herramienta SourceMonitor y los valores umbrales recomendados para algunas de ellas. En la columna característica se recoge la clasificación de la métrica según la característica de calidad asociada: tamaño (TAM), estructura (EST), documentación (DOC) o complejidad (COM).

Tabla 3.11. Conjunto de métricas definido en SourceMonitor

Descripción	Identificador	MinValor	MaxValor	Característica
Nombre proyecto	M0			
Líneas de código	J0			
Nº de sentencias	J1			TAM
% de sentencia condicionales	J2			COM
Nº de llamadas a métodos	J3			
% Líneas de comentarios	J4	8	20	DOC
Nº de clases e interfaces	J5			TAM
Nº de métodos por clase	J6	4	16	EST
Media de sentencias por método	J7	6	12	EST
Máxima complejidad	J10	2	8	
Máxima profundidad de bloques	J12	3	7	
Media de profundidad de bloques	J13	1	2,2	COM
Media de Complejidad	J14	2	4	COM

3.5.3. Operación

La medición de código y clasificación de entidades ha sido realizada por un miembro del tribunal de evaluación, quien previamente ha participado en la formación de los equipos de desarrollo, en concreto, en la enseñanza de programación e ingeniería del software. La instrumentación ha sido realizada en un único ordenador personal en el mes de marzo de 2009. Para cada proyecto se deben identificar las entidades de cada categoría. Posteriormente se calculan las medias aritméticas para cada categoría de las medidas de código descritas en la Tabla 3.11. En la Tabla 3.12 se muestra un ejemplo con los resultados de la medición de una métrica de complejidad sobre un mismo proyecto y respecto a todas las categorías consideradas. La categoría *global*, mostrada en la primera fila de la columna cuya cabecera es *Naturaleza*, hace referencia al cálculo de la métrica sobre todo el proyecto sin hacer ningún tipo de clasificación.

Tomando como referencia las líneas de código (LOC) en la Tabla 3.13 se recoge el tamaño de los códigos analizados dependiendo de la naturaleza del problema. En la inspección realizada, la clasificación de ciertas entidades no se ha podido etiquetar y no se han considerado en el caso de estudio. El porcentaje de clasificación ha sido 76,6 %, siendo la unidad de porcentaje la línea de código.

El conjunto de datos completo generado para su posterior análisis puede ser obtenido en [Lóp12a].

Tabla 3.12. Resultado de la medición de un proyecto

Nombre del proyecto	Naturaleza	Media de la Complejidad
2005-refactorizaciones-xmi	global	2,12
2005-refactorizaciones-xmi	excepción	1
2005-refactorizaciones-xmi	interfaz	1,9
2005-refactorizaciones-xmi	modelo	1,95
2005-refactorizaciones-xmi	control	1,89
2005-refactorizaciones-xmi	prueba	2,59

Tabla 3.13. Tamaño de la muestra, según la naturaleza del problema

Naturaleza de la entidad	LOC
Excepción	8.711
Interfaz gráfico	645.293
Modelo	154.661
Control	182.243
Pruebas	93.730

3.5.4. Análisis

La hipótesis que vamos a estudiar es la siguiente:

H_0 : La métrica M_i se comporta igual en las entidades de código, independientemente de su naturaleza.

H_1 : La métrica M_i se comporta de manera diferente, dependiendo de la naturaleza de las entidades de código.

Donde M_i es una de las métricas de la Tabla 3.11, y la variable dependiente, naturaleza de la entidad, tiene 5 categorías distintas (e_1 exception, e_2 interface, e_3 entity, e_4 control, e_5 test), que definen los niveles del tratamiento. En la réplica no se ha considerado la categoría correspondiente con el esterotipo UML (e_6 utility).

Un test de ANOVA (ANalysis Of VAriance) es el adecuado para contrastar este tipo de hipótesis, tal cuál como se especifica en la Sec. 3.4.4.

3.5.4.1. Resultados observados

En la Sec. 3.5.1 se presentó la hipótesis de interés del caso de estudio, y su refinamiento mediante la hipótesis estadística de análisis de varianza ANOVA.

En la Tabla 3.14 se muestra un resumen de los contrastes de hipótesis llevados a cabo (filas) y de sus resultados al ser aplicados sobre los datos de medición en cada de una de las métricas consideradas (columnas). El valor de cada celda se corresponde con el p-valor. Por otro lado, excepto el conjunto de datos formado por los valores de la

métrica de porcentaje de líneas de comentarios (J4), el resto del conjunto de datos no cumple alguna de las dos precondiciones para aplicar la variante ANOVA de una vía, normalidad y homecedasticidad.

En la Tabla 3.14 el rechazo de la hipótesis H_0 se produce cuando el p-valor observado es menor que 0.05, estos valores aparecen subrayados en la tabla. Los resultados han sido significativos en las 7 métricas consideradas. Por tanto podemos aceptar que la naturaleza de la entidad influye en los valores de las mismas y habría que tener en cuenta dicha información al construir valores umbrales.

Tabla 3.14. p-valores observados, subrayados los no significativos al nivel 0.05

	DOC	ESTRUCTURA			COMPLEJIDAD		
	J4	J1 / J5	J6	J7	J2	J13	J14
Normalidad Shapiro Wilk	<u>0.4281</u>	2.69 e-09	1.152 e-08	0.0007163	2.96 e-06	<u>0.7094</u>	2.459e-06
Igualdad de varianzas Barlett	<u>0.1348</u>	0.000439	0.000433	1.162 e-05	1.40 e-11	0.000216	3.955e-06
ANOVA una vía	<u>5.3 e-12</u>	-	-	-	-	-	-
ANOVA Kruskal Wallis	-	<u>0.000305</u>	<u>0.000101</u>	<u>4.947e-06</u>	<u>0.02264</u>	<u>4.934e-05</u>	<u>0.0001728</u>

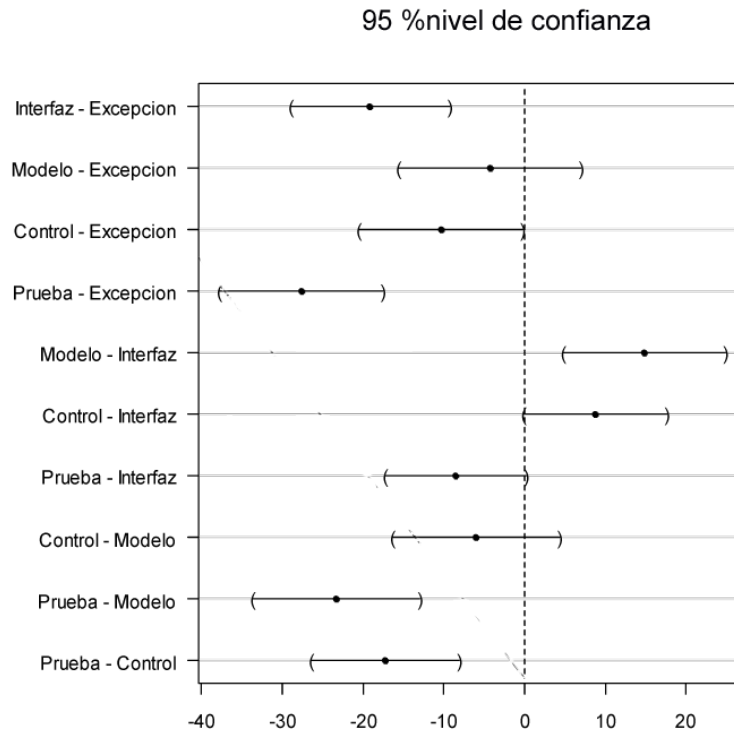
El conjunto de valores de la métrica “porcentaje de líneas de comentarios” (J4), es el único donde se ha podido aplicar el test ANOVA de una vía. En la Figura 3.9 se han calculado los intervalos de confianza de Tukey comparando dos a dos los valores de la media de J4 respecto su naturaleza. El análisis de la salida lleva a las siguientes conclusiones:

- El porcentaje de líneas de comentarios es mayor en e1 excepciones que en el resto de tipos considerados
- El porcentaje de líneas de comentarios en e2 interfaz gráfico es mayor que el de e5 pruebas, y es menor que e3 modelo y e4 controladores
- El porcentaje de líneas de comentarios en e3 modelo es mayor que el de e4 controladores y e5 pruebas
- El porcentaje de líneas de comentarios en e4 controladores es mayor que el de e5 pruebas

3.5.4.2. Valores umbrales dependientes de la naturaleza de la entidad

Como resultado final del caso de estudio, la Tabla 3.15, recoge los valores umbrales para aquellas métricas cuyos resultados son significativos. Los valores umbrales propuestos se han obtenido seleccionando los percentiles 25 (Q1) y 75 (Q3) de cada métrica [FP97].

Basándonos en estos valores umbrales, se propone un escenario posible para utilizar esta información con el fin de identificar medidas anómalas. En la Tabla 3.15 se ha resaltado la columna de la métrica “porcentaje de líneas de comentarios” J4, donde se indican

Figura 3.9. Intervalos de confianza de Tukey para las diferencias del porcentaje de documentación

los valores umbrales para cada uno de los estereotipos considerados $e1$ [36.9 - 56.9], $e2$ [23.45 - 35.05], $e3$ [41.9 - 46], $e4$ [32.05 - 44.75], $e5$ [10.72 - 26.87], mientras en la Tabla 3.11 aparece [8, 20] como valores umbrales recomendados por la herramienta SourceMonitor, para esa métrica.

A partir de esta información podemos destacar tres cosas:

- Los valores umbrales propuestos por la herramienta no sirven en este contexto, ya que sólo el intervalo propuesto en $e5$ está incluido parcialmente en su propuesta
- La precisión (p), entendida como el rango de los valores umbrales, varía según sea el estereotipo considerado. En las clases de modelo ($p_{e3} = 4,1$) es la más alta mientras en las clases de excepción ($p_{e1} = 20$) y pruebas ($p_{e5} = 15,85$) es más baja.
- Los valores umbrales dependen de la naturaleza de la entidad

Análogamente podemos realizar este análisis sobre el resto de mediciones de las tablas.

Tabla 3.15. Métricas de subsistemas: Valores umbrales recomendados según la naturaleza del problema

		DOC	ESTRUCTURA			COMPLEJIDAD		
		J4	J1 / J5	J6	J7	J2	J13	J14
e ₁ exception	Q1	36.900	4.750	1.750	1.000	0.000	0.845	1.000
	Q3	56.900	25.062	4.000	5.785	29.500	2.215	2.670
e ₂ interface	Q1	23.450	44.166	3.960	5.325	8.875	1.835	1.875
	Q3	35.050	66.946	7.405	9.360	13.750	2.645	2.815
e ₃ entity	Q1	41.900	33.111	6.520	2.675	7.050	1.405	1.395
	Q3	46.000	62.189	9.260	4.090	14.000	1.910	2.020
e ₄ control	Q1	32.050	36.197	3.075	5.645	10.300	1.800	1.970
	Q3	44.750	110.153	11.160	9.765	16.800	2.355	2.935
e ₅ test	Q1	10.725	48.625	5.545	4.567	1.375	1.477	1.082
	Q3	26.875	96.305	9.277	9.130	8.575	1.807	1.960

3.6. Caso de estudio 2: Comparación de métodos de clasificación de entidades

En esta sección se presenta un caso de estudio y una réplica, cuyo objetivo es minimizar la amenaza de la Sec. 3.4, relacionada con la subjetividad de la clasificación basada en la naturaleza de entidades de código. Se utiliza la técnica de encuestas a expertos para validar la clasificación de un conjunto de entidades en estereotipos estándar UML.

3.6.1. Definición del caso de estudio: objetivo, hipótesis y variables

El objetivo de este estudio enunciado en formato GQM [BCR94] es el siguiente:

- *Analizar* entidades de código
- *Con el propósito de* evaluar el grado de concordancia de un clasificador automático de las entidades
- *Con respecto a* una clasificación manual
- *Desde el punto de vista* de los investigadores
- *En el contexto* universitario de Ingeniería Informática (I.I.) de la Universidad de Burgos y empresarial.

Por ello, del objetivo principal del estudio, se deriva la siguiente hipótesis:

El método de clasificación automático de entidades de código no es concordante con una clasificación realizada por personas.

La variable independiente es “el método de clasificación de entidades de código” con dos categorías o tratamientos, “automático” o “humano”, luego es una medida con escala nominal.

La variable dependiente es “el resultado de la clasificación de las entidades de código”, que tiene una escala nominal y cuyas categorías se corresponden con los estereotipos estandar UML de la Tabla 3.4.

Hemos considerado, además, otros factores que pueden afectar al resultado de la clasificación:

- Experiencia de los sujetos, medida con escala nominal con las categorías “todos los estudiantes”, “estudiantes expertos”, “todos los profesionales” y “profesionales expertos”.
- Tipos de objetos, con escala nominal cuyas categorías son “entidades de aplicaciones toy” y “entidades de aplicaciones reales”.
- Grado de detalle en la información que describen las entidades de código, medida con escala nominal con las categorías “vista de diseño de la entidad” o “vista de implementación de la entidad”

La experiencia es un factor que se considera en los experimentos en Ingeniería del Software, porque puede afectar al resultado de los tratamientos. En aplicaciones de tipo “toy” es más sencillo comprender la arquitectura global y su relación con la entidad a clasificar. Esta relación entre la entidad y su intención de diseño no es tan sencilla de obtener en una aplicación real. Pensamos que los resultados de la clasificación podrían variar dependiendo de este tipo de información.

El último factor que hemos considerado que puede afectar a los resultados, es la cantidad de información que se tiene de la entidad a clasificar. La vista de diseño contiene la descripción de la entidad basada en los nombres de otras entidades (paquete, clase, métodos) relacionadas y sus tipos asociados. La vista de implementación hace referencia al código de la entidad.

El estadístico utilizado para validar la hipótesis anterior es el coeficiente Kappa Fleiss [Fle71, SB05], que mide el grado de concordancia entre diferentes estrategias de clasificación.

3.6.2. Planificación

El tipo de experimento que hemos realizado es un estudio de casos, de acuerdo a la clasificación dada por Robson [Rob02]. En esta sección se presenta información relativa a los sujetos y objetos del estudio (Sec. 3.6.2.1), diseño de la encuesta utilizada para obtener información sobre la clasificación por humanos (Sec. 3.6.2.2) y por último, cuestiones relacionadas con la instrumentación para llevar a cabo el estudio (Sec. 3.6.2.3).

3.6.2.1. Selección de sujetos y objetos

Se han incorporado dos grupos de sujetos encuestados o evaluadores del estudio. Uno de los grupos se corresponde con alumnos de la asignatura de Diseño y Mantenimiento

del Software II de 5º de Ingeniería Informática en la Universidad de Burgos. El otro grupo estaba formado por profesores y profesionales con los cuales se ha tenido algún tipo de relación profesional. Ambos grupos de sujetos son hispanohablantes. Los profesionales pertenecían a empresas de desarrollo de productos software (CodiceSoftware S.L., Softeca Informática S.L, Centro Regional de Servicios Avanzados S.A) o a departamentos de desarrollo de software de empresas de otros sectores (Grupo Antolín S.A.). Los profesores pertenecían a las Universidades de Burgos y Valladolid (ver Tabla 3.17).

Los objetos del estudio son entidades de código y se extraen de dos aplicaciones disponibles en repositorios de código abierto. Una pequeña aplicación *toy*, llamada *Visor de imágenes* [Lóp] (3 espacios de nombres, 12 clases), y otra aplicación real utilizada para medir la cobertura de pruebas llamada *EclEmma* [Sou] (20 espacios de nombres, 126 clases).

3.6.2.2. Diseño de la encuesta

La encuesta se ha utilizado para recoger la clasificación de entidades de código realizada por personas. Las preguntas de la encuesta están agrupadas en 4 secciones:

- S1: Sección del perfil del inspector, sirve para evaluar la influencia de la experiencia del evaluador en la clasificación. Contiene 6 preguntas donde se consultan aspectos relacionados con la edad, sexo y experiencia.
- S2: Sección de las definiciones de estereotipos UML, sirve para validar las definiciones de las categorías de la clasificación. Contiene 6 preguntas, una por cada estereotipo. Cada definición de estereotipo va acompañada de 6 posibles respuestas, entre las que se debe elegir la definición del mismo.
- S3: Sección de la clasificación de entidades de código de la aplicación *Visor de imágenes* según estereotipos, sirve para validar clasificaciones de entidades extraídas de una aplicación de tipo *toy*, donde el evaluador puede tener una comprensión global y detallada. En esta sección se clasifican dos veces las mismas entidades utilizando dos informaciones diferentes, por un lado la vista de diseño de la entidad con la estructura de espacios de nombres y los nombres de las clases (visión global), y por otro lado, la vista de implementación (visión detallada), con el código de esas clases disponible en un repositorio de código abierto. Las secciones tienen etiquetas S31 y S32, respectivamente, y contienen 5 preguntas cada una.
- S4: Sección de la clasificación de entidades de código *EclEmma*, sirve para validar, sólo con código, las clasificaciones de entidades, extraídas de una aplicación real. Contiene 19 preguntas. El primer bloque de preguntas, solicita clasificar un conjunto de entidades propuestas. El segundo bloque, solicita al sujeto que proporcione una entidad de cada categoría, e1 exception, e2 interface, e3 entity, e4 control, e5 test, e6 utility o e7 desconocida. El objetivo de este segundo bloque, es disponer de entidades de código propuestas por los sujetos para clasificar con el algoritmo, lo que permitirá mejorar la validez del mismo.

En la Tabla 3.16 se presenta un esquema más visual del diseño de la encuesta. El diseño detallado con todas sus preguntas está recogido en el Anexo A.

Tabla 3.16. Esquema de la encuesta

S1		Perfil del inspector
S2		Definiciones estereotipos
S3 Visor	S31	Clasificar entidades descritas globalmente (diseño)
	S32	Clasificar entidades descritas en detalle (código)
S4 EclEmma	B1	Clasificar entidades
	B2	Proponer entidades clasificadas

3.6.2.3. Instrumentación

En el estudio hemos utilizado las herramientas siguientes:

- El algoritmo de clasificación está basado en criterios de convención de nombres de las entidades. Todas las entidades de código tratadas tienen un nombre simple y un nombre cualificado. El nombre cualificado es el nombre simple precedido de los nombres simples de todos los espacios de nombre donde está contenido el elemento. En la Tabla 3.4 se recoge la convención de nombres utilizada, de manera que una entidad se clasificará en una de esas categorías, si contiene en su nombre cualificado alguna de las cadenas referenciadas en la tabla. En caso de conflicto, cuando el nombre cualificado contenga cadenas que correspondan a más de una categoría, prevalece el criterio del nombre simple de la entidad. Aunque en la Tabla 3.4 se presentan los nombres en inglés, se aplica una traducción de los mismos términos en distintos idiomas. Los términos se han traducido al español, porque algunas de las aplicaciones fueron desarrolladas por hispanohablantes.

En [LMC10] se presentó el plugin de Eclipse que ejecuta este algoritmo de manera automática y que es utilizado en este estudio.

- Una plataforma que permita realizar encuestas on-line para obtener las clasificaciones. En este sentido se ha utilizado el portal EncuestaFacil.com, donde se pueden alojar las encuestas y se facilita la posterior recogida de datos (ej. <http://www.encuestafacil.com/RespWeb/Qn.aspx?EID=788250>).

3.6.3. Operación

En este apartado se presenta cómo se ha llevado a cabo el caso de estudio y su réplica.

Se ha realizado el caso de estudio original y una réplica del mismo. En el caso de estudio original, la encuesta se realizó en un laboratorio de prácticas de la asignatura de Diseño y Mantenimiento del Software II, de 5º de Ingeniería Informática en la Universidad de Burgos, durante la sesión de prácticas del 2 de noviembre de 2010. El tiempo que se

dejó para realizar la encuesta fue de 45 minutos. La encuesta fue comenzada por 11 estudiantes y terminada por 8; nos referiremos a este caso como DMSII.

En la réplica, la encuesta fue distribuida mediante un mensaje de correo electrónico a los profesionales. Se dejó como tiempo límite para rellenar la encuesta una semana desde el envío del mensaje (8- 17 de noviembre de 2010). Además, la encuesta podía ser contestada con discontinuidad temporal ya que la plataforma de encuestas on-line permitía guardar y compartir las preguntas contestadas a lo largo de distintas conexiones. La encuesta fue comenzada por 9 profesionales y terminada por 5; nos referiremos a este caso como GIRO.

La Tabla 3.17 contiene una descripción resumida de los sujetos del caso de estudio y su réplica.

Tabla 3.17. Descripción de los casos de estudio

Caso de estudio	N Sujetos	Cualificación	Control
Original DMSII	8 (11)	Alumnos 5º de I.I.	Supervisado
Réplica GIRO	5 (9)	Profesores y profesionales	Sin supervisión

La principal incidencia fue la alta tasa de abandono en la realización de la encuesta, especialmente relevante en la última sección S4, donde se solicitaba sugerir una entidad de cada categoría.

3.6.4. Análisis

Teniendo en cuenta que las variables dependientes tienen escala nominal, el estudio descriptivo de los resultados puede incluir: distribución de frecuencias, moda, diagramas de barras u otros tipos de gráficos equivalentes.

Las preguntas que responderemos se derivan del objetivo de este estudio, propuesto en la Sec. 3.6.1. En función de ellas se plantean las correspondientes hipótesis, como se muestra en la Tabla 3.18.

Tabla 3.18: Preguntas e hipótesis

$H_{0,1}$: ¿Qué grado de concordancia hay en las definiciones de estereotipos?
Continúa en la siguiente página

Tabla 3.18 – continuación de la página anterior

$H_{0,1a}$ No hay concordancia en la definición del estereotipo r en el caso de estudio j , cuando la experiencia es k ($Kappa_{r,j,k} = 0$).

Donde,

$$r \in \{ e_1, \dots, e_7 \},$$

$$j \in \{ \text{DMSII, GIRO} \},$$

$$k \in \{ \text{Expertos, Todos} \}.$$

$H_{0,1b}$ No hay concordancia en la definición de estereotipos en el caso de estudio j , cuando la experiencia es k ($Kappa_{j,k} = 0$).

Donde,

$$j \in \{ \text{DMSII, GIRO} \},$$

$$k \in \{ \text{Expertos, Todos} \}.$$

$H_{0,2a}$: ¿Qué grado de concordancia hay en la clasificación de entidades de código?

$H_{0,2a}$ No hay concordancia en la clasificación de entidades de tipo i , en el estereotipo r , en el caso de estudio j , cuando la experiencia es k ($Kappa_{i,r,j,k} = 0$).

Donde,

$$i \in \{ \text{Visor de imágenes (Toy), EclEmma (Real)} \},$$

$$r \in \{ e_1, \dots, e_7 \},$$

$$j \in \{ \text{DMSII, GIRO} \},$$

$$k \in \{ \text{Expertos, Todos} \}.$$

$H_{0,2b}$ No hay concordancia en la clasificación de entidades de tipo i , en el caso de estudio j , cuando la experiencia es k ($Kappa_{i,j,k} = 0$).

Donde,

$$i \in \{ \text{Visor de imágenes (Toy), EclEmma (Real)} \},$$

$$j \in \{ \text{DMSII, GIRO} \},$$

$$k \in \{ \text{Expertos, Todos} \}.$$

$H_{0,3}$: ¿Qué grado de concordancia hay entre la clasificación automática y la manual?

Continúa en la siguiente página

Tabla 3.18 – continuación de la página anterior**Conjunto de entidades acotado**

$H_{0,31a}$ El método de clasificación automática no es concordante con el humano, para las entidades de tipo i en el estereotipo r , en el caso de estudio j cuando la experiencia es k ($Kappa_{i,r,j,k} = 0$).

Donde,

$i \in \{ \text{Visor de imágenes (Toy), EclEmma (Real)} \}$,

$r \in \{ e_1, \dots, e_7 \}$,

$j \in \{ \text{DMSII, GIRO} \}$,

$k \in \{ \text{Expertos, Todos} \}$.

$H_{0,31b}$ El método de clasificación automática no es concordante con el humano, para las entidades de tipo i en el caso de estudio j cuando la experiencia es k ($Kappa_{i,j,k} = 0$).

Donde,

$i \in \{ \text{Visor de imágenes (Toy), EclEmma (Real)} \}$,

$j \in \{ \text{DMSII, GIRO} \}$,

$k \in \{ \text{Expertos, Todos} \}$.

Conjunto de entidades no acotado

$H_{0,32a}$ El método de clasificación automática no es concordante con el humano, para los estereotipos r ($Kappa_r = 0$).

Donde,

$r \in \{ e_1, \dots, e_7 \}$,

$H_{0,32b}$ El método de clasificación automática no es concordante con el humano.

En cuanto a los contrastes de hipótesis se harán con el estadístico Kappa Fleiss [Fle71, SB05] que mide el grado de concordancia entre diferentes clasificaciones. En este caso mide el grado de concordancia o coincidencia que hay entre un conjunto de sujetos encuestados, cuando clasifican una colección de objetos según una escala nominal. Este estadístico compara las coincidencias observadas con respecto a las que se hubieran podido esperar por causa del azar. Los valores posibles se encuentran en el rango de $[0,1]$, y en [LK77] se propone una posible interpretación de los mismos.

La estimación de Kappa Fleiss la hemos obtenido a partir del software R [OS09], en particular del paquete irr. Como es usual, la hipótesis de no concordancia (nula), se rechazará cuando el p-valor observado sea menor que 0.05.

Tabla 3.19. Interpretación de valores de Kappa

Kappa	Interpretación
< 0	Concordancia pobre
0.0 a 0.20	Concordancia ligera
0.21 a 0.40	Concordancia justa
0.41 a 0.60	Concordancia moderada
0.61 a 0.80	Concordancia sustancial
0.81 a 1.00	Casi una concordancia perfecta

3.6.4.1. Datos observados

Esta sección presenta los datos recogidos en el caso de estudio DMSII y la réplica GIRO, desglosados en tres subsecciones que se corresponden con las de la encuesta (S1, S2, S3):

- En la primera, los datos generales respecto al perfil de los encuestados
- En la segunda, los datos detallados de las respuestas de cada uno de los sujetos encuestados referentes a la clasificación de un conjunto de entidades que se proponía clasificar
- En la tercera se presenta el conjunto de entidades clasificadas que proponen los sujetos encuestados.

En los datos presentados los sujetos encuestados están identificados por un número. El conjunto de datos completo generado para su posterior análisis, está accesible en la dirección [Lóp12a].

Perfil de los sujetos encuestados La información sobre el perfil se extrae de las respuestas a las preguntas de la sección S1 de la encuesta. El perfil de los 11 sujetos encuestados del caso de estudio DMSII, se corresponde con estudiantes de edades comprendidas entre 22 y 27 años (promedio 24,0), todos hombres menos una mujer, cuatro de ellos tienen experiencia profesional y 9/11 alegan tener conocimientos avanzados o muy avanzados en Java o UML. Los identificadores de los sujetos encuestados que reconocen no tener conocimientos avanzados son el 6 y el 11.

El perfil de los 9 sujetos encuestados de la réplica GIRO, se corresponde con profesores y profesionales de edades comprendidas entre 31 y 53 años (promedio 39,2), todos hombres menos una mujer, con experiencia en el sector superior a 5 años, de los que 4/9 alegan tener conocimientos avanzados o muy avanzados en Java o UML. Los identificadores de los sujetos encuestados con conocimientos avanzados son el 2, 7, 8 y 9.

En la Tabla 3.20 se muestra un resumen de los perfiles de los sujetos que han participado en este estudio.

Datos detallados

Tabla 3.20. Resumen de los perfiles de los sujetos en DMSII y GIRO

Caso de estudio	Sujetos	Edad media	% Sujetos con experiencia	% Sujetos con experiencia que realizan toda la encuesta
Original DMSII	11	24,0	81,81 %	66,6 %
Réplica GIRO	9	39,2	44,44 %	60 %

Las Tablas 3.21 y 3.22 presentan los datos detallados de las respuestas del caso de estudio DMSII y su réplica GIRO respectivamente. En las tablas se han sombreado alternativamente las diferentes secciones de la encuesta. La interpretación de los títulos de la columnas de las tablas es la siguiente:

- *Descripción* se refiere a una descripción breve de la pregunta de la encuesta
- *N* se refiere al número de sujetos que responden
- *ID-P* se refiere a la identificación de la pregunta. El identificador es la concatenación de la sección de la encuesta con el número de pregunta
- *evi* número que identifica al sujeto encuestado (evaluador i-esimo)
- *evaut.* resultado de la clasificación automática basada en la convención de nombres de las entidades

Los valores numéricos de las celdas se corresponden con las distintas categorías de la clasificación según la naturaleza expuesto en la Tabla 3.4, y NaN indica que esa pregunta no fue contestada.

Conjunto de entidades propuestas por los sujetos encuestados Las preguntas de la 13 a la 19 de la sección 4, solicitaban al sujeto encuestado que propusiese una entidad de la aplicación *EclEmma* de cada uno de los estereotipos propuestos. El sujeto tenía acceso a las entidades de la aplicación por medio de un repositorio de control de versiones. En la Tabla 3.23 se presenta el conjunto de entidades a clasificar, propuestas por los sujetos de la réplica GIRO. Se destaca que en el caso de estudio DMSII no se recogieron datos al respecto. En el caso de la réplica GIRO la pregunta 13, donde se solicitaba una entidad de tipo *e1 exception*, no fue contestada por nadie.

La interpretación de los títulos de las columnas de la tabla es la siguiente:

- *Nombre* de la entidad de código de la aplicación *EclEmma* propuesta por un sujeto encuestado
- *ID-P* Identificación de pregunta, se concatena la sección de la encuesta y número de pregunta

Tabla 3.21. Resultados detallados del caso de estudio DMSII

Descripción	ID_P	N	ev1	ev2	ev3	ev4	ev5	ev6	ev7	ev8	ev9	ev10	ev11	evaut.
Definición estereotipo "excepcion"	2_1	11	1	1	1	1	1	1	1	1	1	1	1	1
Definición estereotipo "entity"	2_2	11	2	2	6	6	3	2	3	3	3	3	3	3
Definición estereotipo "interface"	2_3	11	3	2	2	2	2	4	2	2	2	2	2	2
Definición estereotipo "test"	2_4	11	5	5	5	5	5	5	5	5	5	5	5	5
Definición estereotipo "utility"	2_5	11	6	3	3	3	6	6	6	6	6	6	6	6
Definición estereotipo "Control"	2_6	11	4	4	4	4	4	3	4	4	4	4	4	4
dominio.Fotografía	3_3	10	7	3	3	3	3	3	3	3	NaN	3	6	3
interfaces.VisorImagenes	3_4	10	2	2	2	2	2	2	4	2	NaN	2	2	2
servicioaccesodatos.FachadaTablaFotografía	3_5	10	3	2	4	4	2	6	2	4	NaN	4	4	4
servicioaccesodatos.Logging	3_6	10	4	4	6	6	4	3	4	6	NaN	6	3	6
dominio.Fotografía	3_7	10	7	3	3	3	3	3	3	3	NaN	3	6	3
interfaces.VisorImagenes	3_8	10	2	2	2	3	2	2	4	2	NaN	2	2	2
servicioaccesodatos.FachadaTablaFotografía	3_9	10	3	2	4	2	4	6	2	4	NaN	4	4	4
servicioaccesodatos.Logging	3_10	10	4	4	6	1	1	3	4	6	NaN	6	3	6
ui.UIPreferences.java	4_3	8	2	2	NaN	7	4	7	3	6	NaN	6	NaN	6
internal.ui.dialogs.MergeSessionsDialog	4_4	8	4	2	NaN	2	3	2	2	2	NaN	3	NaN	2
ui.actions.MergeSessionsAction	4_5	8	6	4	NaN	2	4	4	2	4	NaN	4	NaN	4
core.SessionManager	4_6	8	7	3	NaN	3	5	6	3	4	NaN	4	NaN	4
core.analysis.JavaModelCoverage	4_7	8	7	3	NaN	4	2	7	3	3	NaN	2	NaN	3
core.analysis.JavaElementCoverage	4_8	8	1	3	NaN	4	3	7	3	3	NaN	3	NaN	3
core.SessionManagerTest	4_9	8	5	5	NaN	5	5	5	5	5	NaN	5	NaN	5
ui.coverageview.CoverageView	4_10	8	2	2	NaN	4	2	4	2	2	NaN	2	NaN	2
com.mountainminds.eclemma.ui	4_11	8	2	2	NaN	2	2	7	2	2	NaN	2	NaN	2
com.mountainminds.eclemma.core.test	4_12	8	4	5	NaN	5	5	5	5	5	NaN	5	NaN	5

- *EvHumana* Los valores numéricos de las celdas de esta columna se corresponden con las distintas categorías de la clasificación de las entidades según su naturaleza ($e1 = 1 \dots e7 = 7$).

3.6.4.2. Análisis descriptivo

Las Tablas 3.24 y 3.25 contienen la distribución de frecuencias de las respuestas a cada pregunta de la encuesta del caso de estudio DMSII y su réplica GIRO, respectivamente. La Tabla 3.26 contiene el resumen de la información cuando consideramos conjuntamente el caso de estudio y la réplica. Se ha resaltado en negrita la moda.

El significado de los títulos de las columnas de las tablas es el siguiente:

Tabla 3.22. Resultados detallados de la réplica GIRO

Descripción	ID_P	N	ev1	ev2	ev3	ev4	ev5	ev6	ev7	ev8	ev9	evaut.
Definición estereotipo "excepcion"	2_1	9	1	1	1	1	1	1	1	1	1	1
Definición estereotipo "entity"	2_2	9	3	3	3	3	2	3	3	3	3	3
Definición estereotipo "interface"	2_3	9	2	2	6	4	2	2	2	2	2	2
Definición estereotipo "test"	2_4	9	5	5	5	5	5	5	5	5	5	5
Definición estereotipo "utility"	2_5	9	6	6	4	6	6	6	6	6	6	6
Definición estereotipo "Control"	2_6	9	4	4	3	3	4	4	4	4	4	4
dominio.Fotografía	3_3	8	3	3	NaN	7	4	3	3	3	3	3
interfaces.VisorImagenes	3_4	8	4	2	NaN	2	6	4	2	2	2	2
servicioaccesodatos.FachadaTablaFotografía	3_5	8	6	4	NaN	2	6	2	7	4	3	4
servicioaccesodatos.Logging	3_6	8	6	6	NaN	6	6	6	6	1	6	6
dominio.Fotografía	3_7	8	3	3	NaN	3	3	3	3	3	3	3
interfaces.VisorImagenes	3_8	8	4	2	NaN	2	4	6	2	2	2	2
servicioaccesodatos.FachadaTablaFotografía	3_9	8	6	4	NaN	2	4	4	7	4	4	4
servicioaccesodatos.Logging	3_10	8	6	6	NaN	4	6	1	6	1	6	6
ui.UIPreferences.java	4_3	5	NaN	7	NaN	NaN	7	6	6	3	NaN	6
internal.ui.dialogs.MergeSessionsDialog	4_4	5	NaN	2	NaN	NaN	7	2	2	2	NaN	2
ui.actions.MergeSessionsAction	4_5	6	NaN	4	NaN	NaN	7	6	4	4	NaN	4
core.SessionManager	4_6	5	NaN	4	NaN	NaN	7	4	4	4	NaN	4
core.analysis.JavaModelCoverage	4_7	5	NaN	4	NaN	NaN	7	6	4	3	NaN	3
core.analysis.JavaElementCoverage	4_8	5	NaN	4	NaN	NaN	7	4	3	3	NaN	3
core.SessionManagerTest	4_9	5	NaN	5	NaN	NaN	7	5	5	5	NaN	5
ui.coverageview.CoverageView	4_10	5	NaN	2	NaN	NaN	7	4	2	2	NaN	2
com.mountainminds.elemma.ui	4_11	5	NaN	2	NaN	NaN	7	2	2	2	NaN	2
com.mountainminds.elemma.core.test	4_12	5	NaN	5	NaN	NaN	7	5	5	5	NaN	5

- *Descripción* se refiere descripción breve de la pregunta de la encuesta
- *N* se refiere al número de sujetos que responden
- *ID-P* se refiere a la identificación de pregunta. Se concatena la sección de la encuesta y número de pregunta.
- *Estereotipos UML* (e1...e6) se refiere a la escala nominal de clasificación de las entidades, Tabla 3.4, e7 es la categoría comodín para representar el resto de categorías no consideradas.

En la Tabla 3.24, caso de estudio DMSII, se puede ver que:

- La frecuencia de las modas es bastante alta (cerca al 100%) en la sección de definiciones (S2). Esto quiere decir que los sujetos coincidían en cuanto a las defi-

Tabla 3.23. Entidades propuestas en la réplica GIRO

Nombre de la entidad	ID_P	EvHumana
com.mountainminds.elemma.internal.ui.coverageview	4_14	2
com.mountainminds.elemma.core.ISessionManager.java	4_14	2
com.mountainminds.elemma.ui.launching.CoverageTabGroup.java	4_14	2
com.mountainminds.elemma.internal.ui.coverageview.CoverageView.java	4_14	2
com.mountainminds.elemma.internal.core.analysis.Lines.java	4_15	3
com.mountainminds.elemma.internal.core.analysis.Lines.java	4_15	3
com.mountainminds.elemma.internal.core.analysis.JavaModelCoverage.java	4_15	3
com.mountainminds.elemma.internal.core.SessionManager.java	4_16	4
com.mountainminds.elemma.internal.core.SessionExporter.java	4_16	4
com.mountainminds.elemma.internal.ui.actions.MergeSessionsAction.java	4_16	4
com.mountainminds.elemma.core.AllEclEmmaCoreTests.java	4_17	5
com.mountainminds.elemma.internal.core.analysis.CounterTest.java	4_17	5
com.mountainminds.elemma.internal.core.SessionManagerTest.java	4_17	5
com.mountainminds.elemma.core.CoverageTools.java	4_18	6
com.mountainminds.elemma.internal.core.instr.InstrMarker.java	4_18	6

niciones de las categorías de estereotipos, excepto en la definición del estereotipo “entity”, que los sujetos identifican con las categorías e3 entity, e2 interface y e6 utility, ordenadas de mayor a menor frecuencia.

- La frecuencia de las modas en la clasificación de entidades del Visor de Imágenes (S3), está por debajo del 50 % en las entidades relativas al espacio de nombres “servicioaccesodatos”.
- En la clasificación de entidades EclEmma (S4), la frecuencia de las modas está por encima del 60 % en la mayoría de los casos, excepto en “ui.Preferences”, “core.analysis.JavaModelCoverage” y “core.analysis.JavaElementCoverage”. Las dos primeras pertenecen a la categoría de control e4 y la última, a la de entity e3.

En la Tabla 3.25, correspondiente a la réplica GIRO, se puede ver que:

- La frecuencia de las modas está por encima del 70 % en todos los casos. Eso quiere decir que en la definición de los conceptos los sujetos coincidieron en un alto porcentaje.
- La frecuencia de las modas en la clasificación de entidades del Visor de Imágenes (S3) está por encima del 50 % excepto en la entidad “servicioaccesodatos.FachadaTablaFotografia” que los sujetos identifican con las categorías e2, e3, e4, e6 y e7.
- En la clasificación de entidades EclEmma (S4), la frecuencia de las modas está por encima del 60 % en la mayoría de los casos, excepto en “ui.Preferences”,

Tabla 3.24. Resumen del caso de estudio DMSII

Descripción	ID_P	N	e1	e2	e3	e4	e5	e6	e7
Definición estereotipo "excepcion"	2_1	11	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Definición estereotipo "entity"	2_2	11	0.0%	27.3%	54.5%	0.0%	0.0%	18.2%	0.0%
Definición estereotipo "interface"	2_3	11	0.00%	81.00%	9.10%	9.10%	8.3%	0.0%	0.0%
Definición estereotipo "test"	2_4	11	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%
Definición estereotipo "utility"	2_5	11	0.0%	0.0%	27.3%	0.0%	0.0%	72.7%	0.0%
Definición estereotipo "Control"	2_6	11	0.0%	0.0%	9.1%	90.9%	0.0%	0.0%	0.0%
dominio.Fotografía	3_3	10	0.0%	0.0%	80.0%	0.0%	0.0%	10.0%	10.0%
interfaces.VisorImagenes	3_4	10	0.0%	90.0%	0.0%	10.0%	0.0%	0.0%	0.0%
servicioaccesodatos.FachadaTablaFotografia	3_5	10	0.0%	30.0%	10.0%	50.0%	0.0%	10.0%	0.0%
servicioaccesodatos.Logging	3_6	10	0.0%	0.0%	20.0%	40.0%	0.0%	40.0%	0.0%
dominio.Fotografía	3_7	10	0.0%	0.0%	80.0%	0.0%	0.0%	10.0%	10.0%
interfaces.VisorImagenes	3_8	10	0.0%	80.0%	10.0%	10.0%	0.0%	0.0%	0.0%
servicioaccesodatos.FachadaTablaFotografia	3_9	10	0.0%	30.0%	10.0%	50.0%	0.0%	10.0%	0.0%
servicioaccesodatos.Logging	3_10	10	20.0%	0.0%	20.0%	30.0%	0.0%	30.0%	0.0%
ui.UIPreferences.java	4_3	8	0.0%	25.0%	12.5%	12.5%	0.0%	25.0%	25.0%
internal.ui.dialogs.MergeSessionsDialog	4_4	8	0.0%	62.5%	25.0%	12.5%	0.0%	0.0%	0.0%
ui.actions.MergeSessionsAction	4_5	8	0.0%	25.0%	0.0%	62.5%	0.0%	12.5%	0.0%
core.SessionManager	4_6	8	0.0%	0.0%	37.5%	25.0%	12.5%	12.5%	12.5%
core.analysis.JavaModelCoverage	4_7	8	0.0%	25.0%	37.5%	12.5%	0.0%	0.0%	25.0%
core.analysis.JavaElementCoverage	4_8	8	12.5%	0.0%	62.5%	12.5%	0.0%	0.0%	12.5%
core.SessionManagerTest	4_9	8	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%
ui.coverageview.CoverageView	4_10	8	0.0%	75.0%	0.0%	25.0%	0.0%	0.0%	0.0%
com.mountainminds.eclEmma.ui	4_11	8	0.0%	87.50%	0.0%	0.0%	0.0%	0.0%	12.5%
com.mountainminds.eclEmma.core.test	4_12	8	0.0%	0.0%	0.0%	12.5%	87.5%	0.0%	0.0%

“core.analysis.JavaModelCoverage” y “core.analysis.JavaElementCoverage”, como sucedía en el caso de estudio DMSII.

En la Tabla 3.26, cuando se estudian conjuntamente los resultados de DMSII y GIRO, se puede ver que:

- En la sección de definiciones (S2), la frecuencia de las modas está por encima del 80 % en todos los casos.
- En la sección de la clasificación de entidades del Visor de Imágenes (S3) la frecuencia de las modas está por encima del 50 % en la mayoría de los casos, excepto para las entidades “servicioaccesodatos.FachadaTablaFotografia” y “servicioaccesodatos.Logging”, que se corresponde con las entidades en las que, en DMSII y GIRO, la clasificación tiene una moda < 50 %.
- En la sección de la clasificación de entidades EclEmma (S4), la frecuencia de las modas está por encima del 50 % en la mayoría de los casos, excepto para las entidades “ui.Preferences”, “core.SessionManager” y “core.analysis.JavaModelCoverage”, tal como sucedía, tanto en DMSII como en GIRO.

Tabla 3.25. Resumen de la réplica GIRO

Descripción	ID_P	N	e1	e2	e3	e4	e5	e6	e7
Definición estereotipo "excepcion"	2_1	9	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Definición estereotipo "entity"	2_2	9	0.0%	11.1%	88.9%	0.0%	0.0%	0.0%	0.0%
Definición estereotipo "interface"	2_3	9	0.0%	77.8%	0.0%	11.1%	0.0%	11.1%	0.0%
Definición estereotipo "test"	2_4	9	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%
Definición estereotipo "utility"	2_5	9	0.0%	0.0%	0.0%	11.1%	0.00%	88.9%	0.0%
Definición estereotipo "Control"	2_6	9	0.0%	22.2%	0.0%	77.8%	0.0%	0.0%	0.0%
dominio.Fotografia	3_3	8	0.0%	0.0%	75.0%	12.5%	0.0%	0.0%	12.5%
interfaces.VisorImagenes	3_4	8	0.0%	62.5%	0.0%	25.0%	0.0%	12.5%	0.0%
servicioaccesodatos.FachadaTablaFotografia	3_5	8	0.0%	25.0%	12.5%	25.0%	0.0%	25.0%	12.5%
servicioaccesodatos.Logging	3_6	8	12.5%	0.0%	0.0%	0.0%	0.0%	87.5%	0.0%
dominio.Fotografia	3_7	8	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
interfaces.VisorImagenes	3_8	8	0.0%	62.5%	0.0%	25.0%	0.0%	12.5%	0.0%
servicioaccesodatos.FachadaTablaFotografia	3_9	8	0.0%	12.5%	0.0%	62.5%	0.0%	12.5%	12.5%
servicioaccesodatos.Logging	3_10	8	25.0%	0.0%	0.0%	12.5%	0.0%	62.5%	0.0%
ui.UIPreferences.java	4_3	5	0.0%	0.0%	20.0%	0.0%	0.0%	40.0%	40.0%
internal.ui.dialogs.MergeSessionsDialog	4_4	5	0.0%	80.0%	0.0%	0.0%	0.0%	0.0%	20.0%
ui.actions.MergeSessionsAction	4_5	6	0.0%	0.0%	0.0%	60.0%	0.0%	20.0%	20.0%
core.SessionManager	4_6	5	0.0%	0.0%	0.0%	80.0%	0.0%	0.0%	20.0%
core.analysis.JavaModelCoverage	4_7	5	0.0%	0.0%	20.0%	40.0%	0.0%	20.0%	20.0%
core.analysis.JavaElementCoverage	4_8	5	0.0%	0.0%	40.0%	40.0%	0.0%	0.0%	20.0%
core.SessionManagerTest	4_9	5	0.0%	0.0%	0.0%	0.0%	80.0%	0.0%	20.0%
ui.coverageview.CoverageView	4_10	5	0.0%	60.0%	0.0%	20.0%	0.0%	0.0%	20.0%
com.mountainminds.elemma.ui	4_11	5	0.0%	80.0%	0.0%	0.0%	0.0%	0.0%	20.0%
com.mountainminds.elemma.core.test	4_12	5	0.0%	0.0%	0.0%	0.0%	0.0%	80.0%	20.0%

3.6.4.3. Estudio de las hipótesis sobre concordancia en las clasificaciones

En esta sección nos ocuparemos de contrastar las hipótesis relativas a este estudio. No hemos considerado grupo de expertos frente a no expertos, porque este último grupo tenía muy pocos sujetos, por ello estudiamos expertos frente a todos. Se considera expertos a aquellos sujetos que, en la sección de perfil de la encuesta, han contestado que tienen unos conocimientos avanzados o muy avanzados de Java o UML.

Además, cuando la clasificación se hace en los objetos del tipo de aplicación *toy*, *Visor de imágenes*, se estudian dos clasificaciones, S31 y S32, que corresponden a las mismas entidades cuando se proporciona distinta información para hacer la clasificación (descriptiva o detallada).

El conjunto de entidades acotado se corresponde con entidades propuestas por los investigadores, mientras que el *no acotado* se corresponden con entidades propuestas por los sujetos encuestados.

Estudio de la concordancia en la definición de conceptos

Tabla 3.26. Resumen del caso de estudio DMSII y la réplica GIRO

Descripción	ID_P	N	e1	e2	e3	e4	e5	e6	e7
Definición estereotipo "excepcion"	2_1	20	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Definición estereotipo "entity"	2_2	20	0.0%	20.0%	70.0%	0.0%	0.0%	10.0%	0.0%
Definición estereotipo "interface"	2_3	21	0.0%	80.0%	5.0%	10.0%	0.0%	5.0%	0.0%
Definición estereotipo "test"	2_4	20	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%
Definición estereotipo "utility"	2_5	20	0.0%	0.0%	15.0%	5.0%	0.0%	80.0%	0.0%
Definición estereotipo "Control"	2_6	20	0.0%	10.0%	5.0%	85.0%	0.0%	0.0%	0.0%
dominio.Fotografia	3_3	18	0.0%	0.0%	77.8%	5.6%	0.0%	5.6%	11.1%
interfaces.VisorImagenes	3_4	18	0.0%	77.8%	0.0%	16.7%	0.0%	5.6%	0.0%
servicioaccesodatos.FachadaTablaFotografia	3_5	18	0.0%	27.8%	11.1%	38.9%	0.0%	16.7%	5.6%
servicioaccesodatos.Logging	3_6	18	5.6%	0.0%	11.1%	22.2%	0.0%	61.1%	0.0%
dominio.Fotografia	3_7	18	0.0%	0.0%	88.9%	0.0%	0.0%	5.6%	5.6%
interfaces.VisorImagenes	3_8	18	0.0%	72.2%	5.6%	16.7%	0.0%	5.6%	0.0%
servicioaccesodatos.FachadaTablaFotografia	3_9	18	0.0%	22.2%	5.6%	55.6%	0.0%	11.1%	5.6%
servicioaccesodatos.Logging	3_10	18	22.2%	0.0%	11.1%	22.2%	0.0%	44.4%	0.0%
ui.UIPreferences.java	4_3	13	0.0%	15.4%	15.4%	7.7%	0.0%	30.8%	30.8%
internal.ui.dialogs.MergeSessionsDialog	4_4	13	0.0%	69.2%	15.4%	7.7%	0.0%	0.0%	7.7%
ui.actions.MergeSessionsAction	4_5	14	0.0%	15.4%	0.0%	61.5%	0.0%	15.4%	7.70%
core.SessionManager	4_6	13	0.0%	0.0%	23.1%	46.2%	7.7%	7.7%	15.4%
core.analysis.JavaModelCoverage	4_7	13	0.0%	15.4%	30.8%	23.1%	0.0%	7.7%	23.1%
core.analysis.JavaElementCoverage	4_8	13	7.7%	0.0%	53.8%	23.1%	0.0%	0.0%	15.4%
core.SessionManagerTest	4_9	13	0.0%	0.0%	0.0%	0.0%	92.3%	0.0%	7.7%
ui.coverageview.CoverageView	4_10	13	0.0%	69.2%	0.0%	23.1%	0.0%	0.0%	7.7%
com.mountainminds.elemma.ui	4_11	13	0.0%	84.6%	0.0%	0.0%	0.0%	0.0%	15.4%
com.mountainminds.elemma.core.test	4_12	13	0.0%	0.0%	0.0%	7.7%	53.8%	30.8%	7.7%

En este apartado se estudia las hipótesis $H_{0,1,i}$ de la Tabla 3.18 utilizando el estadístico Kappa, interpretado como en la Tabla 3.19.

En la Tabla 3.27 se presentan los resultados de analizar los datos de la sección S2 de la encuesta. Este conjunto de preguntas se refieren a la identificación de conceptos de estereotipos UML. Se han resaltado en negrita los valores de Kappa con concordancias perfectas, sustanciales o moderadas y los p-valores significativos. Además, tanto para el caso de estudio como para la réplica, se ha considerado la variable experiencia del evaluador.

Tabla 3.27. Concordancia en la definición de conceptos

- / j,k	S2		
j=DMSII, k=todos	r	$Kappa_{r,j,k}$	p.value
	e1	1.000	0.000
	e2	0.572	0.000
	e3	0.193	0.000
	e4	0.782	0.000
	e5	1.000	0.000
	e6	0.505	0.000
	e7	-	-
	Questions = 6 Raters = 11 $Kappa_{j,k} = 0.676$ p-value = 0		
j=DMSII, k=expertos	r	$Kappa_{r,j,k}$	p.value
	e1	1.000	0.000
	e2	0.662	0.000
	e3	0.233	0.001
	e4	1.000	0.000
	e5	1.000	0.000
	e6	0.413	0.000
	e7		
	Questions = 6 Raters = 9 $Kappa_{j,k} = 0.722$ p-value = 0		
j=GIRO, k=todos	r	$Kappa_{r,j,k}$	p.value
	e1	1.000	0.000
	e2	0.596	0.000
	e3	0.662	0.000
	e4	0.500	0.000
	e5	1.000	0.000
	e6	0.733	0.000
	e7		
	Questions = 6 Raters = 9 $Kappa_{j,k} = 0.75$ p-value = 0		
j=GIRO, k=expertos	r	$Kappa_{r,j,k}$	p.value
	e1	1	0
	e2	1	0
	e3	1	0
	e4	1	0
	e5	1	0
	e6	1	0
	e7	-	-
	Questions = 6 Raters = 4 $Kappa_{j,k} = 1$ p-value = 0		

Las hipótesis a estudiar con los resultados se muestran a continuación.

$H_{0,1a}$ No hay concordancia en la definición del estereotipo r en el caso de estudio j , cuando la experiencia es k ($Kappa_{r,j,k} = 0$).

Donde,

$$r \in \{ e_1, \dots, e_7 \},$$

$$j \in \{ DMSII, GIRO \},$$

$$k \in \{ Expertos, Todos \}.$$

El grado de concordancia cuando se consideran todas las categorías ha sido el siguiente:

- En el caso de estudio DMSII se aprecia una concordancia muy débil en la definición de conceptos e_3 entity ($Kappa_{e_3,DMSII,Todos} = 0.193$) y más moderada en e_4 control ($Kappa_{e_4,DMSII,Todos} = 0.572$) y e_6 utility ($Kappa_{e_6,DMSII,Todos} = 0.505$).
- En GIRO la concordancia menor es para las definiciones, son moderadas y pertenecen a las categorías e_2 interface ($Kappa_{e_2,GIRO,Todos} = 0.596$) y e_4 control ($Kappa_{e_4,GIRO,Todos} = 0.500$).

$H_{0,1b}$ No hay concordancia en la definición de estereotipos en el caso de estudio j , cuando la experiencia es k ($Kappa_{j,k} = 0$).

Donde,

$$j \in \{ DMSII, GIRO \},$$

$$k \in \{ Expertos, Todos \}.$$

A partir de los resultados observados podemos ver que hay una concordancia sustancial en cuanto a las definiciones de estereotipos de UML. Los resultados han sido similares cuando se han considerado todos los encuestados y cuando se ha condicionado por la experiencia, aunque el grado de concordancia mejora en este último caso:

- En DMSII ligeramente $Kappa_{DMSII,Expertos} = 0.722$ frente a $Kappa_{DMSII,Todos} = 0.676$.
- En el caso de GIRO se obtiene concordancia perfecta $Kappa_{GIRO,Expertos} = 1$ y $Kappa_{GIRO,Todos} = 1$.

Podemos concluir que:

El grado de concordancia en la definición de conceptos por diferentes sujetos, es de moderado a perfecto en el caso de los expertos de GIRO, y mejor que la observada en DMSII.

Estudio de la concordancia en la clasificación de entidades

En este apartado se estudia las hipótesis $H_{0,2,i}$ de la Tabla 3.18 utilizando el estadístico Kappa, interpretado como en la Tabla 3.19.

$H_{0,2a}$ No hay concordancia en la clasificación de objetos de tipo i , en el estereotipo r , en el caso de estudio j , cuando la experiencia es k ($Kappa_{i,r,j,k} = 0$).

Donde,

$$\begin{aligned}
i &\in \{ \text{Visor de imágenes (Toy), EclEmma (Real)} \}, \\
r &\in \{ e_1, \dots, e_7 \}, \\
j &\in \{ \text{DMSII, GIRO} \}, \\
k &\in \{ \text{Expertos, Todos} \}.
\end{aligned}$$

En la Tabla 3.28 se resumen los resultados de este test de hipótesis, al analizar los datos de las secciones S31, S32 y S4 de la encuesta presentados en las Tablas 3.21 y 3.22. Se han resaltado en negrita los valores de Kappa con concordancias perfectas, sustanciales o moderadas y los p-valores que permitirán decidir si rechazamos o no la hipótesis nula. Además, tanto para el caso original como para la réplica, hemos realizado dos análisis, uno sin considerar la variable experiencia en Java o UML (todos) y otro, considerándola (expertos).

A la vista de los resultados podemos decir que la concordancia, tanto en definir como en clasificar, no es muy intensa cuando estudiamos a todos los sujetos sin considerar su experiencia, en el experimento ($Kappa_{S31,DMSII,Todos} = 0.332$, $Kappa_{S32,DMSII,Todos} = 0.252$, $Kappa_{S4,DMSII,Todos} = 0.329$) en la réplica (GIRO $Kappa_{S31,GIRO,Todos} = 0.288$, $Kappa_{S32,GIRO,Todos} = 0.41$, $Kappa_{S4,GIRO,Todos} = 0.269$).

Examinando los resultados del caso de estudio DMSII para cada pregunta y para cada objeto examinado (Tabla 3.28), podemos observar que:

- Para la clasificación realizada por todos y basada en el diseño (S31), hubo concordancia entre los sujetos para las entidades e2, e3 y e4, ordenadas de mayor a menor. Para las entidades e1 y e5, no se pudo calcular Kappa porque ningún sujeto clasificó en esas categorías y en e6 y e7 la concordancia fue no significativa.
- Para la clasificación realizada por los expertos y basada en el diseño (S31), hubo concordancia entre los sujetos para las entidades e3, e2 y e6, ordenadas de mayor a menor. Para las entidades e1 y e5, no se pudo calcular Kappa porque ningún sujeto clasificó en esas categorías, y en e4 y e7 la concordancia fue no significativa.
- Para la clasificación realizada por todos y basada en la implementación (S32), hubo concordancia entre los sujetos para las entidades e2 y e3, ordenadas de mayor a menor. Para la entidad e5 no se pudo calcular Kappa porque ningún sujeto clasificó en esa categoría, y en las restantes la concordancia fue no significativa.
- Para la clasificación realizada por los expertos y basada en la implementación (S32), hubo concordancia entre los sujetos para las entidades e3, e2 y e6, ordenadas de mayor a menor, como en la hecha basándose en el diseño (S31). Para la entidad e5, no se pudo calcular Kappa porque ningún sujeto clasificó en esas categorías, y en las otras tres entidades la concordancia fue no significativa.
- Para la clasificación realizada por todos en las entidades de EclEmma (S4), hubo concordancia entre los sujetos para las entidades e5, e2 y e3, ordenadas de mayor a menor. En el resto la concordancia fue no significativa.
- Para la clasificación realizada por expertos en las entidades de EclEmma (S4), los resultados fueron los mismos que en el caso anterior (todos).

Tabla 3.28. Concordancia en la clasificación de entidades

i / j,k	i=S31			i=S32			i=S4		
	r	$Kappa_{i,r,j,k}$	p.value	r	$Kappa_{i,r,j,k}$	p.value	r	$Kappa_{i,r,j,k}$	p.value
j=DMSII, k= todos	e1	-	-	e1	0.064	0.338	e1	-0.013	0.832
	e2	0.603	0.000	e2	0.485	0.000	e2	0.405	0.000
	e3	0.429	0.000	e3	0.339	0.000	e3	0.208	0.000
	e4	0.141	0.000	e4	0.124	0.097	e4	0.085	0.156
	e5	-	-	e5	-	-	e5	0.844	0.000
	e6	0.085	0.254	e6	0.010	0.898	e6	0.023	0.706
	e7	0.026	0.731	e7	0.026	0.731	e7	-0.006	0.914
	Questions = 4 Raters = 10 $Kappa_{i,j,k} = 0.332$ p-value = 4.22e-15			Questions = 4 Raters = 10 $Kappa_{i,j,k} = 0.252$ p-value = 2.68e-10			Questions = 10 Raters = 8 $Kappa_{i,j,k} = 0.329$ p-value = 0		
j=DMSII, k=expertos	e1	-	-	e1	0.086	0.364	e1	-0.014	0.834
	e2	0.543	0.000	e2	0.404	0.000	e2	0.482	0.000
	e3	0.667	0.000	e3	0.536	0.000	e3	0.256	0.000
	e4	0.139	0.142	e4	0.095	0.313	e4	0.028	0.687
	e5	-	-	e5	-	-	e5	0.821	0.000
	e6	0.347	0.000	e6	0.212	0.025	e6	0.071	0.301
	e7	-0.032	0.733	e7	-0.032	0.733	e7	-0.045	0.516
	Questions = 4 Raters = 8 $Kappa_{i,j,k} = 0.412$ p-value = 2.22e-14			Questions = 4 Raters = 8 $Kappa_{i,j,k} = 0.3$ p-value = 2.58e-09			Questions = 10 Raters = 7 $Kappa_{i,j,k} = 0.364$ p-value = 0		
j=GIRO, k=todos	e1	-0.032	0.733	e1	0.086	0.364	e1	-	-
	e2	0.295	0.002	e2	0.355	0.000	e2	0.592	0.000
	e3	0.504	0.000	e3	1.000	0.000	e3	0.049	0.625
	e4	0.050	0.599	e4	0.190	0.044	e4	0.287	0.004
	e5	-	-	e5	-	-	e5	0.702	0.000
	e6	0.460	0.000	e6	0.242	0.010	e6	0.049	0.625
	e7	0.067	0.480	e7	-0.032	0.733	e7	-0.224	0.025
	Questions = 4 Raters = 8 $Kappa_{i,j,k} = 0.288$ p-value = 3.65e-09			Questions = 4 Raters = 8 $Kappa_{i,j,k} = 0.41$ p-value = 0			Questions = 10 Raters = 5 $Kappa_{i,j,k} = 0.269$ p-value = 1.94e-08		
j=GIRO, k=expertos	e1	-0.067	0.744	e1	-0.067	0.744	e1	-	-
	e2	1.000	0.000	e2	1.000	0.000	e2	1.000	0.000
	e3	0.709	0.010	e3	1.000	0.000	e3	0.135	0.461
	e4	0.238	0.243	e4	0.590	0.004	e4	0.683	0.000
	e5	-	-	e5	-	-	e5	1.000	0.000
	e6	0.590	0.004	e6	0.590	0.004	e6	-0.034	0.850
	e7	-0.067	0.744	e7	-0.067	0.744	e7	-0.034	0.850
	Questions = 4 Raters = 4 $Kappa_{i,j,k} = 0.573$ p-value = 2.84e-08			Questions = 4 Raters = 4 $Kappa_{i,j,k} = 0.686$ p-value = 9.42e-12			Questions = 10 Raters = 3 $Kappa_{i,j,k} = 0.693$ p-value = 2.17e-12		

Examinando los resultados de la réplica GIRO para cada pregunta y para cada objeto examinado (Tabla 3.28), podemos observar que:

- Para la clasificación realizada por todos y basada en el diseño (S31), hubo concordancia entre los sujetos para las entidades e3, e6 y e2, ordenadas de mayor a menor. Para la entidad e5, no se pudo calcular Kappa porque ningún sujeto clasificó en esas categorías, y en el resto la concordancia fue no significativa.
- Para la clasificación realizada por expertos y basada en el diseño (S31), hubo concordancia entre los sujetos para las entidades e2, e3 y e6, ordenadas de mayor a menor, mejorando sustancialmente el grado de concordancia obtenido por todos. Para la entidad e5, no se pudo calcular Kappa porque ningún sujeto clasificó en esas categorías, y en e1, e4 la concordancia fue no significativa.
- Para la clasificación realizada por todos y basada en la implementación (S32), hubo concordancia entre los sujetos para las entidades e2, e3 y e6, ordenadas de mayor a menor. Para la entidad e5, no se pudo calcular Kappa porque ningún sujeto clasificó en esas categorías, y en e1 y e4 la concordancia fue no significativa.
- Para la clasificación realizada por expertos y basada en la implementación (S32), hubo concordancia entre los sujetos para las entidades e2, e3, e4 y e6, ordenadas de mayor a menor, mejorando sustancialmente el grado de concordancia obtenido por todos. Para la entidad e5, no se pudo calcular Kappa porque ningún sujeto clasificó en esas categorías, y en el resto la concordancia fue no significativa.
- Para la clasificación realizada por todos en las entidades de EclEmma (S4), hubo concordancia entre los sujetos para las entidades e5, e2 y e4, ordenadas de mayor a menor. En el caso de e7 se obtuvo un grado de concordancia pobre y en el resto, la concordancia fue no significativa.
- Para la clasificación realizada por los expertos en las entidades de EclEmma (S4), hubo concordancia entre los sujetos para las entidades e5, e2 y e4, ordenadas de mayor a menor, mejorando sustancialmente el grado de concordancia obtenido por todos. En el resto la concordancia fue no significativa.

Viendo la Tabla 3.29, que resume los resultados significativos para la hipótesis $H_{0,2a}$, podemos concluir que:

- Los expertos presentan una coincidencia similar al clasificar con menos detalle (S31) o con más detalle (S32) en ambos casos de estudio, DMSII y GIRO.
- Al clasificar las entidades de EclEmma (S4) existe coincidencia entre “todos” y “expertos”.

La otra hipótesis para estudiar la concordancia en la clasificación de entidades, por diferentes sujetos, es la siguiente:

$H_{0,2b}$ No hay concordancia en la clasificación de objetos de tipo i , en el caso de estudio j , cuando la experiencia es k ($Kappa_{i,j,k} = 0$).

Donde,

$i \in \{ \text{Visor de imágenes (Toy), EclEmma (Real)} \}$,

Tabla 3.29. Resumen de resultados significativos para la concordancia en clasificar, por estereotipos, objetos y experiencia $H_{0,2a}$

	S31		S32		S4	
	Todos	Expertos	Todos	Expertos	Todos	Expertos
DMSII	e2, e3, e4	e3, e2, e6	e2, e3	e3, e2, e6	e5, e2, e3	e5, e2, e3
GIRO	e3, e6, e2	e2, e3, e6	e3, e2, e6	e2, e3, e4, e6	e5, e2, e4	e5, e2, e4

$$j \in \{ \text{DMSII, GIRO} \} ,$$

$$k \in \{ \text{Expertos, Todos} \} .$$

Los resultados observados al estudiar la concordancia por caso de estudio (Tabla 3.28) y grupo de experiencia son los siguientes:

- Los expertos de DMSII tuvieron un grado de concordancia entre justa y moderada ($Kappa_{S31} = 0.412$, $Kappa_{S32} = 0.3$, $Kappa_{S4} = 0.364$).
- Los expertos de la réplica (GIRO) obtuvieron resultados más favorables, pues hay un grado de concordancia entre moderada y sustancial ($Kappa_{S31} = 0.573$, $Kappa_{S32} = 0.686$, $Kappa_{S4} = 0.693$).

Los resultados observados cuando se consideraron “todos” los sujetos fueron peores que en el grupo de “expertos”, en ambos estudios. Luego parece que influye el caso de estudio y la experiencia.

Estudio de la concordancia entre evaluación humana y automática

En este caso, las hipótesis de interés las hemos desglosado respecto del conjunto de entidades acotado y no acotado. El conjunto de entidades acotado se corresponde con las entidades propuestas a todos los sujetos, procedentes de *Visor de imágenes* y *EclEmma*, mientras que el *no acotado* se corresponde con entidades propuestas por sujetos de la réplica GIRO, procedentes de *EclEmma*. Los sujetos del caso de estudio DMSII no propusieron entidades *no acotadas*.

Respecto al conjunto acotado, disponemos de entidades de código que han sido clasificadas automáticamente (con el algoritmo de convención de nombres, automatizado con un plugin de Eclipse [LMC10]) y manualmente por los sujetos encuestados. Como se dispone de más de una clasificación humana para cada entidad de código, hemos elegido la moda como valor representativo de todas ellas, dado que la escala de la clasificación es nominal (ver Tabla 3.26). La moda ha sido calculada considerando las variables dependientes experiencia y tipo de objeto.

Las hipótesis las hemos enunciado como:

Conjunto de entidades acotado

$H_{0,31a}$ El método de clasificación automática no es concordante con el humano, para las entidades de tipo i en el estereotipo r , en el caso de estudio j cuando la expe-

riencia es k ($Kappa_{i,r,j,k} = 0$).

Donde,
 $i \in \{ \text{Visor de imágenes, EclEmma} \}$,
 $r \in \{ e_1, \dots, e_7 \}$,
 $j \in \{ \text{DMSII, GIRO} \}$,
 $k \in \{ \text{Expertos, Todos} \}$.

$H_{0,31_b}$ El método de clasificación automática no es concordante con el humano, para las entidades de tipo i en el caso de estudio j cuando la experiencia es k ($Kappa_{i,j,k} = 0$).

Donde,
 $i \in \{ \text{Visor de imágenes (S32), EclEmma} \}$,
 $j \in \{ \text{DMSII, GIRO} \}$,
 $k \in \{ \text{Expertos, Todos} \}$.

Conjunto de entidades no acotado En los datos observados no se dispone de un conjunto de entidades acotadas para el caso de estudio DMSII. Además en la réplica GIRO el conjunto de entidades no acotadas es proporcionado sólo por expertos. Por esta razón no podemos considerar las variables $j \in \{ \text{DMSII, GIRO} \}$ y $k \in \{ \text{Expertos, Todos} \}$ en las siguientes hipótesis.

$H_{0,32_a}$ El método de clasificación automática no es concordante con el humano, para los estereotipos r ($Kappa_r = 0$).

Donde,
 $r \in \{ e_1, \dots, e_7 \}$,

$H_{0,32_b}$ El método de clasificación automática no es concordante con el humano.

En la Tabla 3.30 se pueden observar los valores de Kappa y los p-valores para cada caso de estudio al estudiar la experiencia en las hipótesis $H_{0,31_a}$ y $H_{0,31_b}$. Se han resaltado en negrita los valores de Kappa con concordancias perfectas, sustanciales o moderadas y los p-valores que permitirán decidir si rechazamos o no la hipótesis nula.

Tabla 3.30. Resultados de concordancia de evaluación humana vs. automática respecto a entidades acotadas

i / j,k	i=S32			i=S4			
	r	$Kappa_{i,r,j,k}$	p.value	r	$Kappa_{i,r,j,k}$	p.value	
j=DMSII, k=todos	e1	-	-	e1	-	-	
	e2	1.000	0.046	e2	0.780	0.014	
	e3	1.000	0.046	e3	0.733	0.020	
	e4	0.467	0.351	e4	0.608	0.055	
	e5	-	-	e5	1.000	0.002	
	e6	-	-	e6	-0.053	0.868	
	e7	-	-	e7	-	-	
	Questions = 4 Raters = 2 $Kappa_{i,j,k} = 0.652$ p-value = 0.0308			Questions = 10 Raters = 2 $Kappa_{i,j,k} = 0.733$ p-value = 2.82e-05			
	j=DMSII, k=expertos	e1	-	-	e1	-	-
		e2	1.000	0.046	e2	0.780	0.014
e3		1.000	0.046	e3	0.733	0.020	
e4		0.467	0.351	e4	0.608	0.055	
e5		-	-	e5	1.000	0.002	
e6		-0.143	0.775	e6	-0.053	0.868	
e7		-	-	e7	-	-	
Questions = 4 Raters = 2 $Kappa_{i,j,k} = 0.652$ p-value = 0.0308			Questions = 10 Raters = 2 $Kappa_{i,j,k} = 0.733$ p-value = 2.82e-05				
j=GIRO, k=todos		e1	-	-	e1	-	-
		e2	1.000	0.046	e2	1.000	0.002
	e3	1.000	0.046	e3	-0.111	0.725	
	e4	1.000	0.046	e4	0.524	0.098	
	e5	-	-	e5	1.000	0.002	
	e6	1.000	0.046	e6	-0.053	0.868	
	e7	-	-	e7	-0.053	0.868	
	Questions = 4 Raters = 2 $Kappa_{i,j,k} = 1$ p-value = 0.000532			Questions = 10 Raters = 2 $Kappa_{i,j,k} = 0.608$ p-value = 0.000279			
	j=GIRO, k=expertos	e1	-	-	e1		
		e2	1.000	0.046	e2	1.000	0.002
e3		1.000	0.046	e3	0.608	0.055	
e4		1.000	0.046	e4	0.733	0.020	
e5		-	-	e5	1.000	0.002	
e6		1.000	0.046	e6	-0.053	0.868	
e7		-	-	e7	-0.053	0.868	
Questions = 4 Raters = 2 $Kappa_{i,j,k} = 1$ p-value = 0.000532			Questions = 10 Raters = 2 $Kappa_{i,j,k} = 0.744$ p-value = 4.5e-06				

Examinando los resultados de la Tabla 3.30 se puede observar que:

- En el caso de estudio DMSII la variable experiencia es independiente respecto a la concordancia de las clasificaciones automática y humana. Los diferentes valores de Kappa coinciden para cada estereotipo, en ambos tipos de objetos (S32 y S4).
- En el caso de estudio DMSII y objeto Visor de Imágenes (S32) la concordancia entre la clasificación automática y humana es perfecta para los estereotipos e2 y e3 y no se obtienen resultados significativos para e4 y e6. De manera global, sin considerar el estereotipo de la entidad, la concordancia de ambas clasificaciones es sustancial ($Kappa_{S32,DMSII,Todos} = 0.652$, $Kappa_{S32,DMSII,Expertos} = 0.652$).
- En el caso de estudio DMSII y objeto EclEmma (S4) la concordancia entre la clasificación automática y humana es perfecta en e5, sustancial en e2, e3 y e4. De manera global, sin considerar el estereotipo de la entidad, la concordancia de ambas clasificaciones es sustancial ($Kappa_{S4,DMSII,Todos} = 0.733$, $Kappa_{S4,DMSII,Expertos} = 0.733$).
- En el caso de estudio GIRO la variable experiencia influye levemente respecto a la concordancia de clasificación entre el método automático y el manual. En ambos casos la concordancia es sustancial pero se observa que a mayor experiencia, mayor concordancia con el método automático ($Kappa_{S4,GIRO,Todos} = 0.608$ vs. $Kappa_{S4,GIRO,Expertos} = 0.744$).
- En el caso de estudio GIRO y objeto Visor de Imágenes (S32) la concordancia entre la clasificación automática y humana es perfecta para los estereotipos e2, e3, e4 y e6, independientemente de la experiencia. Por tanto de manera global, sin considerar el estereotipo de la entidad, la concordancia de ambas clasificaciones es perfecta ($Kappa_{S32,GIRO,Todos} = 1$ y $Kappa_{S32,GIRO,Expertos} = 1$).
- En el caso de estudio GIRO, objeto EclEmma (S4) y sin considerar la experiencia (Todos) la concordancia entre la clasificación automática y humana es perfecta en e2 y e5; y sustancial en e4. Cuando se considera la experiencia, las concordancias de las clasificaciones se incrementa un poco, en e2 y e5 es perfecta; y en e3 y e4 es sustancial. De manera global, sin considerar el estereotipo de la entidad, la concordancia de ambas clasificaciones es sustancial ($Kappa_{S4,GIRO,Todos} = 0.608$, $Kappa_{S4,GIRO,Expertos} = 0.744$).

Respecto a las hipótesis $H_{0,32a}$ y $H_{0,32b}$ del conjunto de entidades *no acotado*, la Tabla 3.31 presenta los análisis basados en Kappa. A las entidades clasificadas (Tabla 3.23), propuestas por los sujetos encuestados en la réplica GIRO, se aplica el algoritmo automático de clasificación obteniendo de nuevo dos clasificaciones (humana y automática). De manera global, cuando no consideramos el estereotipo (hipótesis $H_{0,32b}$), los resultados son favorables a la existencia de concordancia sustancial en ambas clasificaciones (Kappa=0.656). En cuanto a la hipótesis $H_{0,32a}$, los resultados están a favor de una concordancia perfecta en clasificaciones de entidades con estereotipo e5, sustancial en e2 y e3 y moderada en e4. En e1 no se contaba con ninguna propuesta y en e6 los resultados son no significativos.

Tabla 3.31. Resumen del grado de concordancia entre la evaluación humana y la automática, respecto a entidades no acotadas, por expertos de GIRO

r	$Kappa_{EclEmma, GIRO, r, Expertos}$	p.value
e1	-	-
e2	0.814	0.002
e3	0.659	0.011
e4	0.441	0.088
e5	1.000	0.000
e6	0.071	0.782

Questions = 15
Raters = 2
 $Kappa_{EclEmma, GIRO, Expertos} = 0.656$
p-value = **1.69e-06**

3.6.4.4. Amenazas a la validez

La validez de construcción del caso de estudio, entendida como las variables que han sido correctamente medidas, está amenazada por el hecho de que la clasificación de entidades en estereotipos UML no es ortogonal cuando se aplica sobre entidades de código reales.

La validez interna del caso de estudio, entendida como la causalidad de nuestras conclusiones, está afectada por las siguientes amenazas:

- El abandono de algunos evaluadores seleccionados para realizar la encuesta redujo el número de evaluaciones, eso produce indeterminación cuando no se rechaza H_0 , pues la potencia de los test empeora
- La realización de las encuestas sin supervisión

Desde la perspectiva de la validez externa, referida a cuántos de nuestros resultados se pueden generalizar en otras circunstancias, se han identificado las siguientes amenazas:

- La extensión de los resultados está limitada por los proyectos utilizados y el lenguaje de programación de los mismos
- La formación de los encuestados no es homogénea, y ello puede condicionar los resultados de la encuesta

3.6.5. Conclusiones sobre la clasificación automática

El doble objetivo de este estudio es, por un lado validar que los expertos coinciden en la clasificación de entidades de código en estereotipos UML. Por otro lado, que la clasificación realizada con el algoritmo coincide con la de los expertos. Este objetivo es de interés porque las técnicas de clasificación de entidades de código son útiles para

mejorar las técnicas de detección de defectos de diseño. Si los expertos coinciden, entonces la clasificación puede utilizarse para definir las reglas de detección de defectos. Si el algoritmo clasifica coincidiendo con los expertos, la detección automática se facilita, pues la clasificación humana en sistemas reales sería una tarea ingente.

Las conclusiones se van a referir a los tres análisis parciales de concordancia que hay en este estudio:

1. Estudio del grado de concordancia en la definición de estereotipos UML, realizada por sujetos. Los resultados indican que: hay concordancia en las definiciones de estereotipos UML por parte de los sujetos, con cierta independencia de la experiencia, aunque llega a ser perfecta en el caso de los expertos de GIRO. Por ello la experiencia es un factor que influye en los resultados. Se concluye que podemos utilizar la clasificación de la entidad en la definición de estrategias de detección de defectos.
2. Estudio del grado de concordancia en la clasificación de entidades de código, realizada por sujetos. Los resultados obtenidos nos indican que:
 - La concordancia en la clasificación es mayor cuando los sujetos son expertos. La influencia de la experiencia en la clasificación de las entidades de Visor imágenes (S3) proporciona resultados diferentes en DMSII y en GIRO. Los expertos de DMSII obtienen un grado de concordancia mayor en la clasificación cuando se hace a partir de información de diseño. Cuando se estudia el grado de concordancia al clasificar todas las entidades, los expertos obtienen mejores resultados. En ambos casos, se mejoran los resultados obtenidos al considerar a todos los sujetos. Basta mirar los coeficientes Kappa de expertos en ambos grupos (Tabla 3.28). Luego la experiencia es un factor a tener en cuenta.
 - Los sujetos expertos coinciden más cuando identifican entidades de tipo e2 interface y e5 test, y discrepan moderadamente cuando son de tipo e6 utilidad y e4 control (Tabla 3.28).
 - El grado de concordancia al clasificar entidades de tipo e3 entity varía sustancialmente dependiendo de la aplicación, Visor de imágenes (aplicación toy) o EclEmma (aplicación real). De hecho es mayor en la primera. Esta variación puede deberse a que el dominio de la aplicación real puede necesitar más tiempo y más conocimientos para ser comprendido. Luego influye la aplicación.
 - Existe poca variación en los resultados de la clasificación, dependiendo de la información que se proporciona, información de la entidad en vista de diseño o en vista de implementación. Por lo tanto es más económico proporcionar el diseño para clasificar la entidad. Los expertos de GIRO mejoran el grado de concordancia cuando usan la implementación.

En el caso de S32 y expertos, se tiene concordancia sustancial en e2 y e3, moderada en e4 y e6. Cuando se trata de S4 y expertos, existe concordancia sustancial o perfecta en e2, e4 y e5. Esta diferencia puede deberse a que

los expertos de GIRO son profesionales, más habituados a leer código que los expertos de DMSII (estudiantes) y son capaces de comprender mejor las entidades descritas con código, y por ello, las clasifican mejor cuando están así descritas.

3. Estudio del grado de concordancia entre la clasificación automática y la humana, realizada por sujetos. Los resultados nos indican:
 - El tipo de entidad y el estereotipo considerado influyen ligeramente en los resultados de la concordancia, aunque en la mayoría de los casos la concordancia se mantiene por lo menos moderada.
 - El grado de concordancia entre la clasificación automática y la humana, en las entidades propuestas por los sujetos del estudio fue sustancial.
 - La experiencia influye en el grado de concordancia entre clasificación automática y humana, los sujetos de GIRO, que son los más expertos, obtienen coeficientes de concordancia mejores que los de DMSII, aunque en ambos casos los resultados son al menos de concordancia moderada.

Por todo ello, podemos concluir que, la clasificación automática podría sustituir a la clasificación humana.

3.7. Conclusiones de la experimentación

Los casos de estudio presentados en las Sec. 3.4 y 3.5 han confirmado que la naturaleza de las entidades de código influyen en las mediciones de las mismas. Como aplicación de este resultado, hemos propuesto valores umbrales para cada métrica, condicionados por la naturaleza o estereotipo de la entidad de código, que permitirán identificar medidas anómalas en contextos de desarrollo similares. La principal amenaza de estos estudios experimentales es el método automático de clasificación de las entidades de código en estereotipos UML.

A partir de los resultados obtenidos en el caso de estudio de la Sec. 3.6 se puede concluir que los dos métodos de clasificación, el automático y el guiado por un inspector, tienen una coincidencia perfecta a sustancial para los seis estereotipos UML considerados. Aunque el grado de concordancia disminuye en los estereotipos de control y utilidad como ocurría con la concordancia entre los propios expertos humanos. En el conjunto de entidades no acotado, la coincidencia entre clasificación manual y automática fue significativa en todos los estereotipos, excepto en e6, del que no había datos.

En la Tabla 3.32 se muestra un resumen descriptivo de las conclusiones de los casos de estudio respecto a sus hipótesis principales y sus resultados significativos.

La extrapolación de los resultados de estos casos de estudios a una empresa que desarrolle aplicaciones necesita crear un histórico de medidas clasificadas en estereotipos de UML para poder generar sus valores umbrales aplicando los estadísticos de agrupación basados en percentiles [FP97]. Además el equipo de desarrollo es el más adecuado para

Tabla 3.32. Resumen de las conclusiones de los casos de estudio

Caso de estudio	Hipótesis	%Significativos	Sección
Plugings de Eclipse	H_0 : La métrica M_i se comporta igual en las entidades de código, independientemente de su naturaleza.	87% (34/39)	3.4
Réplica Trabajos Fin de Carrera		100% (7/7)	3.5
Original GIRO	$H_{0,32a}$ El método de clasificación automática no es concordante con el humano, para los estereotipos r ($Kappa_r = 0$). Donde, $r \in \{ e_1, \dots, e_7 \}$,	80% (4/5)	3.6
Réplica DMSII		Sin datos	3.6

realizar la clasificación de las entidades y generar sus tablas de convención de nombres. En definitiva, el proceso seguido en este caso de estudio es perfectamente aplicable a una empresa que quiera dirigir sus decisiones apoyándose en las métricas, pues nuestra hipótesis de trabajo le puede aportar información relevante para tomar decisiones.

Aunque se han realizado dos réplicas, creemos que se necesitan más. Las nuevas réplicas se pueden definir con otros conjuntos de proyectos seleccionados a partir de factores externos similares: porcentaje de actividad, popularidad, estado de desarrollo y área funcional. Con ello se pretende refinar y comparar los valores umbrales propuestos y probar si son dependientes de estos factores. Además, pensamos que para validar nuestra propuesta de proceso de medición, podemos incorporar nuevos conjuntos de métricas [Moo05].

Se necesita validar mediante experimentación el proceso de medición propuesto. Para ello, queremos estudiar si se obtienen mejores resultados en la detección de entidades anómalas respecto de un proceso de medición que no considere la naturaleza de la entidad.

PROCESO DINÁMICO DE GESTIÓN DE DEFECTOS

La capacidad de gestionar defectos de diseño en el software puede ayudar a mejorar la calidad y a hacer más productivas las tareas de desarrollo y de mantenimiento. En sistemas reales, donde existe una gran cantidad de entidades de código, un enfoque de detección de defectos de diseño basado únicamente en la intuición y experiencia humana es inabordable. La detección de defectos debería ser asistida por un proceso que estructure una inspección semi-automática. En el estudio realizado en el Cap. 2 se encontraron herramientas que implementan la detección de algunos de los defectos de diseño catalogados. No obstante, cuando un inspector realiza una inspección de defectos de diseño, existe una componente de subjetividad que debe ser gestionada en el proceso. Las herramientas disponibles actualmente no tienen en cuenta lo anterior, es decir, no proporcionan al auditor una funcionalidad para adaptar las reglas o heurísticas de detección a su contexto de aplicación.

El objetivo de este capítulo es triple y sirve para estructurar sus secciones.

El primero, Sec. 4.1, es definir un proceso genérico y dinámico de gestión de defectos basado en métricas de código, en la naturaleza o intención de diseño de la entidad y en clasificadores binarios. Se pretende describir un proceso con el suficiente detalle para que pueda ser repetido o adaptado como sea necesario para algún defecto particular u otra actividad similar de mantenimiento.

El segundo objetivo es aplicar el proceso en un contexto concreto (ver Sec.4.2), sobre defectos relacionados con características de tamaño, acoplamiento y complejidad: *God Class* (ver Sec.4.3), *Data Class* (ver Sec. 4.4) y *Feature Envy* (ver Sec.4.5).

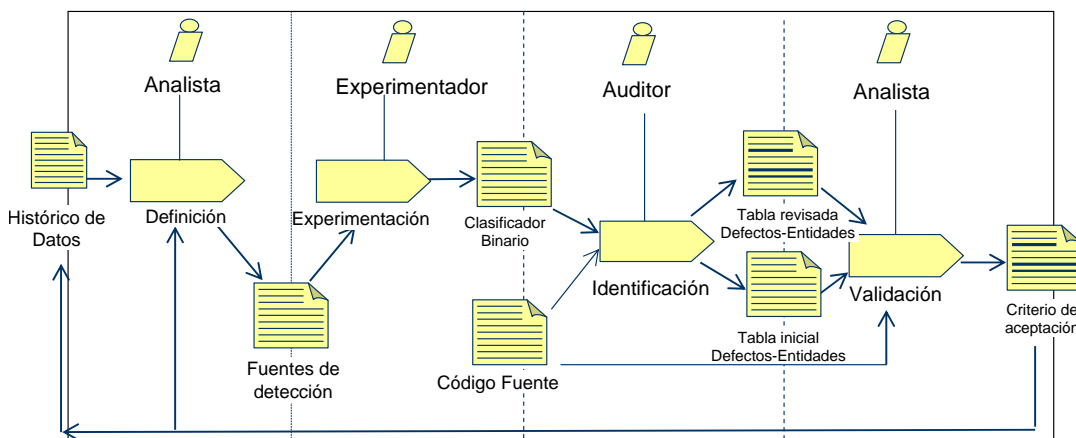
El tercer objetivo es realizar un estudio empírico para validar la eficiencia de los métodos de detección en los tres defectos de diseño antes mencionados (*God Class*, *Data Class* y *Feature Envy*) al incorporar en la detección información sobre la naturaleza de la entidad, Sec 4.6 en adelante. Este estudio persigue, por un lado especificar de manera concreta las dos actividades novedosas propuestas en el proceso (experimentación y

validación). Por otro lado, validar las hipótesis para el cual fue diseñado dicho caso de estudio, es decir, comprobar si la naturaleza de la entidad, modelada como estereotipo UML, mejora la predicción de defectos de diseño.

4.1. Proceso genérico de gestión de defectos de diseño

En esta sección se describe un proceso dinámico de gestión de defectos de diseño mediante métricas. Para la descripción de los elementos del proceso se utiliza la notación *Software Process Engineering Metamodel Specification (SPEM) en su versión 2.0* [OMG06]. En la Figura 4.1 se presenta una visión global del proceso cuyos elementos (participantes, tareas y productos) se describen en las siguientes subsecciones. La principal fortaleza del proceso es la incorporación de la tarea de validación que gestiona la subjetividad asociada a la identificación de defectos. Esta tarea se basa en la aplicación del aprendizaje supervisado. El auditor, mediante técnicas de clasificación de minería de datos [MGF07], incorpora al aprendizaje la subjetividad y las características particulares del contexto.

Figura 4.1. Proceso dinámico de gestión de defectos



4.1.1. Participantes

En el proceso se identifican los siguientes roles:

- *Analista* utiliza el *histórico de datos* para definir el defecto de código, caracterizando las entidades de código con alguna de las *fuentes de detección*. En nuestro contexto particular las *fuentes* serán, las métricas de código e información sobre la naturaleza o intención de diseño de la entidad, obteniendo su tipo de acuerdo a estereotipos estándar UML [JBR00, AN05]. Además, en otra fase del proceso a partir de las *tablas inicial y revisada de defectos-entidades* realiza su validación.

- *Experimentador*, partiendo de la definición del analista, es el encargado de generar con técnicas de minería de datos *clasificadores binarios* para la detección de defectos.
- *Auditor*, partiendo del *clasificador* (árbol de decisión) proporcionado por el experimentador y del *código fuente* del sistema software a evaluar, es el encargado de identificar entidades anómalas en el sistema.

El perfil profesional relacionado con los roles de analista y auditor, es la Ingeniería de Desarrollo de Software, por ello ambos roles los podría desempeñar la misma persona. El rol de experimentador debería tener un perfil relacionado con la Ingeniería del Software Empírica.

4.1.2. Tareas

- *Definición*, el objetivo de esta tarea es seleccionar el conjunto de entidades que sirven para definir el defecto. Los defectos se definen como un conjunto de entidades de código caracterizadas con unos atributos, denominados *fuentes de detección*. Los atributos se corresponden con: valores de varias medidas de código, la naturaleza de diseño de la entidad, medida inicialmente como escala nominal con los estereotipos UML, y el etiquetado binario “con defecto” o “sin defecto”.
- *Experimentación*, el objetivo de esta tarea es explorar distintas ejecuciones de los algoritmos de clasificación para proponer un clasificador. El clasificador propuesto es un árbol de decisión con un conjunto de reglas definidas como una expresión que contiene atributos de las entidades de código y de operadores relacionales que predicen si una entidad tiene un defecto.
- *Identificación*, el objetivo de esta tarea es buscar entidades (capas, paquetes, subsistemas, clases, métodos) con defectos de diseño sobre el código fuente de un sistema software. En esta tarea, se revisa la *tabla inicial de defectos-entidades* buscando falsos positivos y falsos negativos.
- *Validación*, el objetivo de esta tarea es validar la *tabla revisada de defectos-entidades*. Con los resultados de esta validación se permite retroalimentar de nuevo la definición de defectos generando un nuevo *clasificador*.

La tarea de validación utiliza medidas de eficiencia de los clasificadores *Recuperación*, *Precisión* y *F-Measure* descritas en [WFH11]. Se utiliza *F-Measure* como indicador de condición de parada del proceso, por ser la medida cuya interpretación relaciona la *Recuperación* y la *Precisión*. Un valor mínimo aceptable para *F-Measure* es 0.66, que se obtiene asignando a la precisión, la moda de probabilidad industrial propuesta en [Fag02].

4.1.3. Productos

En el proceso se identifican los siguientes productos:

- *Código fuente*, cuya calidad necesita ser evaluada respecto a los defectos de diseño. Es el producto de entrada en la tarea de identificación.
- *Clasificador binario*, es un árbol de decisión cuyos nodos se corresponden con información utilizada para la detección y las ramas son expresiones lógicas formadas por un operador relacional y un valor numérico. Es el producto de salida obtenido en la tarea de experimentación. Los árboles de decisión son muy similares a los sistemas de predicción basados en reglas.
- *Histórico de datos, tabla inicial de defectos-entidades y tabla revisada de defectos-entidades*, son diferentes conjuntos de instancias (o entidades de código) utilizadas en la clasificación. El *histórico* se corresponden con las instancias utilizadas para generar el clasificador. La *tabla inicial* es el conjunto de instancias predichas automáticamente, la *tabla revisada* es el conjunto de instancias con defecto inspeccionadas por el auditor detectando falsos positivos y falsos negativos en la *tabla inicial*.
- *Criterio de aceptación*, es el valor asignado a *F-Measure* que permite tomar la decisión de retroalimentar la definición del defecto. Se acompaña de toda la información utilizada para calcular *F-Measure*: tabla de contingencia o confusión, *Recuperación* y *Precisión*.

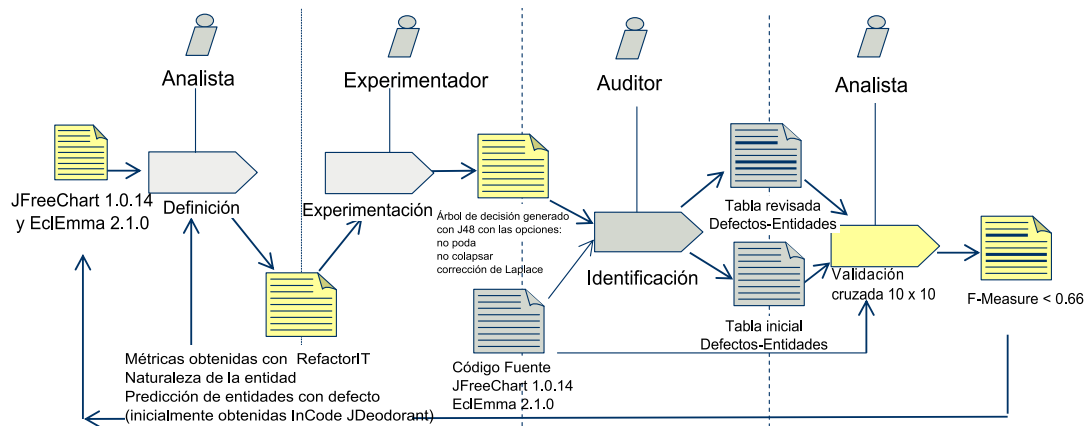
4.2. Aplicación del proceso

En esta sección se describe la aplicación del proceso en un contexto concreto. En la Sec. 4.2.1 se describe la obtención de los productos de entrada al proceso: el histórico de datos iniciales en el que basar la definición del defecto y el código fuente del sistema a evaluar. En la Sec. 4.2.2 se describe en detalle la tarea de validación aplicada y su relación con las técnicas de aprendizaje supervisado. Por último, en la Sec. 4.2.3, se proporciona la plantilla descriptiva que será utilizada en la presentación de cada defecto concreto. La información de esta sección es común en la aplicación del proceso sobre cada defecto seleccionado: *God Class* (ver Sec.4.3), *Data Class* (ver Sec. 4.4) y *Feature Envy* (ver Sec.4.5).

Como resumen del contexto de aplicación del proceso en la Fig. 4.2 se muestran los elementos concretos que son explicados con más detalle en las siguientes secciones.

4.2.1. Datos de entrada

El *histórico de datos*, base para aplicar este proceso, ha sido obtenido a partir de entidades de código de las aplicaciones de código abierto JFreeChart 1.0.14 y EclEmma 2.1.0 (disponible en [Lóp12a]). La clasificación inicial de entidades con defecto para formar el histórico se obtiene con herramientas de otros autores considerados como expertos. En sucesivas iteraciones del proceso, las definiciones de los defectos se pueden retroalimentar de nuevos atributos o nuevas instancias.

Figura 4.2. Contexto de aplicación del proceso de gestión de defectos

Inicialmente los atributos o fuentes de detección utilizadas para determinar la definición de los defectos son:

- Métricas estructurales de código obtenidas con la herramienta RefactorIT (ver Tabla 3.2).
- Métrica nominal semántica relacionada con la naturaleza de la entidad (ver Tabla 3.4) calculada con el Algoritmo 1.
- Predicción binaria de defectos en entidades de código obtenida por medio de las herramientas inCode [Int08] y JDeodorant [JDe11].

4.2.2. Validación de las decisiones del analista

En esta fase se utiliza una técnica de aprendizaje supervisada o predictiva. Los algoritmos supervisados estiman la función f que mejor asocia un conjunto de datos X (variables independientes) con un conjunto de datos Y (variables dependientes), dado un conjunto anterior de observaciones (datos a priori). En este contexto, las variables independientes se corresponden con los valores de las medidas (estructurales o semánticas) de las entidades de código y la variable dependiente, con la predicción binaria del defecto en la entidad. Los datos a priori se corresponden inicialmente con el histórico base y posteriormente con la retroalimentación de éste mediante las tablas revisadas defecto-entidades.

Esta forma de trabajar se conoce como aprendizaje supervisado y se desarrolla en dos fases: fase de entrenamiento o supervisión y fase de prueba. La fase de entrenamiento se corresponde con la tarea de experimentación y la fase de prueba con la tarea de identificación (ver Sec.4.1.2). El método de aprendizaje seleccionado es el árbol de decisión, que usa las reglas de predicción para clasificar las instancias (entidades de código) según sus atributos. Este método requiere clasificaciones discretas y predefinidas. El

uso primario de este método es predecir modelos que sean apropiados para cualquier clasificación o técnicas de regresión.

Los árboles de decisión son utilizados fundamentalmente para clasificación y segmentación, consisten en una serie de comprobaciones que van separando el problema, siguiendo la técnica “divide y vencerás”, hasta llegar a las hojas del árbol que determinan la clase o grupo a la que pertenece el dato. Existen muchísimas técnicas para inducir árboles de decisión, siendo la más conocida el algoritmo C4.5 de Quinlan [Qui92], cuya implementación en la herramienta de minería de datos Weka se denomina J48 [WFH11].

Para estimar el error o determinar el rendimiento del algoritmo se usa una validación cruzada 10-veces (del inglés *10-fold cross validation*) sobre un conjunto de datos iniciales que contiene instancias clasificadas. Este procedimiento divide los datos disponibles en 10 subconjuntos de instancias y toma sucesivamente un conjunto diferente como conjunto de prueba usando el resto de instancias (de los otros subconjuntos) como conjunto de histórico. De esta forma, el algoritmo se ejecuta 10 veces aprendiendo en cada ejecución con un 90 % de las instancias del histórico y se prueba con el 10 % de instancias restantes.

Para evaluar los posibles resultados cuando se predice una clasificación con dos alternativas (*tiene defecto/no tiene defecto*), se utiliza una tabla como la Tabla 4.1. En ésta, la diagonal principal recoge las predicciones acertadas, la otra diagonal los dos posibles errores:

- Falso positivo (FP): se produce cuando la salida se predice incorrectamente como “sí” (o positivo) cuando realmente es “no” (negativo)
- Falso negativo (FN): se produce cuando la salida se predice incorrectamente como “no” (negativo) cuando realmente es “sí” (positivo).

Las medidas seleccionadas para medir la bondad del clasificador obtenido en la fase de entrenamiento son:

$$Precision = \frac{TP}{TP+FP}$$

$$Recuperacion = \frac{TP}{TP+FN}$$

A partir de ellas se puede calcular una única medida conocida como F-Measure utilizada en la tarea de validación 4.1.2 (media armónica o media de ratios), y se define como:

$$F - Measure = \frac{2*Recuperacion*Precision}{Recuperacion+Precision} = \frac{2*TP}{2*TP+FP+FN}$$

F-Measure es la medida que se va a utilizar en el criterio de aceptación, en concreto $F - Measure < 0,66$.

Tabla 4.1. Salidas del proceso de aprendizaje en una predicción binaria

		Clase predicha	
		sí	no
Clase real	sí	Positivos ciertos (TP)	Falsos negativos (FN)
	no	Falsos positivos (FP)	Negativos ciertos (TN)

4.2.3. Plantilla de descripción del defecto

Para documentar y describir el resultado de la aplicación del proceso expuesto para cada defecto de código se utilizará la plantilla que se muestra en la Tabla 4.2.

4.3. Aplicación del proceso a God Class

En esta sección se aplica el proceso de gestión de defectos de diseño definido en la Sec.4.1 para el defecto *God Class*. Se utilizan como productos de entrada (histórico de datos y códigos fuentes) los presentados en la Sec.4.2.1 y como criterio de aceptación el expuesto en la Sec. 4.2.2.

Las subsecciones se corresponden con los campos de la plantilla propuesta para presentar la aplicación del proceso para un defecto (ver Sec.4.2.3).

4.3.1. Clasificación y motivación

En los buenos diseños orientados a objetos la lógica del sistema está distribuida uniformemente entre las clases de los niveles superiores. En [Rie96] se define una *God Class* como un objeto que controla demasiados objetos en el sistema y ha crecido más allá de toda lógica para convertirse en la clase que lo hace todo. A veces se corresponde con una mala aplicación del patrón de diseño mediador [GHJV95]. Este problema de diseño puede ser parcialmente asimilado con el defecto *Large Class* definido por Fowler [FBB⁺99]. También es conocido como el antipatrón *The Blob* en [BMMM98].

4.3.2. Heurísticas para su detección

Marinescu [Mar02b] propone basarse en tres características para definir su estrategia de detección:

1. La clase accede a muchos datos de otras clases. Se define la métrica de acceso a datos foráneos (*ATFD Access To Foreign Data*).
2. La complejidad funcional de la clase es muy alta.
3. La cohesión es baja. Se define una métrica de cohesión (*TCC Tight Class Cohesion*).

Tabla 4.2. Descripción de la aplicación del proceso definido en 4.1

Clasificación y motivación	Descripción textual del defecto y clasificaciones conocidas
Heurísticas para su detección	Fuentes de detección y reglas de detección de otros autores
Definición	<p>Conjunto de datos resumido de entrenamiento. Cada fila se corresponde con una entidad de código. Las columnas se organizan como sigue:</p> <ul style="list-style-type: none"> ▪ 1...n valores de métricas obtenidos con RefactorIt ▪ n+1 la naturaleza de la entidad calculada con el algoritmo de clasificación expuesto en el Cap. 3 ▪ n+2 el valor predicho del defecto (true o false) por inCode o JDeodorant <p>Los conjuntos completos de datos no se incluyen en el texto por su excesivo tamaño pero son accesibles en [Lóp12a].</p>
Experimentación	<p>Árbol de decisión obtenido por el conjunto de entrenamiento. Son diagramas de construcciones lógicas, muy similares a los sistemas de predicción basados en reglas utilizados por otros autores en la detección de defectos (ver Cap. 2). Los árboles se obtienen con la aplicación del algoritmo J48. En [CHCK12] se recomienda aplicar el algoritmo de clasificación J48 vs C45 para conjuntos de datos desequilibrados con las opciones: no poda, no colapsar y corrección de Laplace. Para disminuir el tamaño en la presentación de estos árboles en esta memoria, se realizará una poda manual y se indicarán los atributos afectados.</p>
Validación	Análisis de los resultados de la validación cruzada. En este apartado se presentarán objetos, sujetos y medidas de rendimiento del clasificador, resultado del proceso de validación cruzada.

En la definición de las reglas se utilizan lo que el autor denomina filtros estadísticos basados en gráficos de cajas (*box plot*). Esta técnica sirve para fijar umbrales que detectan valores anormales (*outlier*) en las medidas de las entidades.

La estrategia expresada como conjunto de reglas es la siguiente:

```
GodClasses := ((ATFD > FEW) and (WMC >= VERY HIGH) and (TCC < ONE THIRD))
```

Por su parte, Moha et.al [MGDL10] utilizan un lenguaje específico de dominio para especificar defectos y el método DECOR define el proceso completo de especificación,

detección y visualización. DECOR define la detección de *God Class* buscando: clases que son controladoras (regla *ControllerClass*), clases grandes (regla *LargeClass*) o de cohesión débil (regla *LowCohesion*), y clases que se asocian con una o muchas clases de datos (regla *DataClass*). A raíz de la heurística de Riel [Rie96], una clase controladora se puede identificar por su nombre o por los nombres de sus métodos, que debe contener términos como: *Process*, *Control*... La cohesión de la clase se caracteriza utilizando la métrica LCOM. Finalmente, una clase se considera grande si su número de métodos y campos (medidos con las métricas NMD y NAD) son muy altos (mayor que los valores medios de las clases en el programa). Las *Data Class* son las clases con más del 90% de métodos de acceso. A continuación se presenta el conjunto de reglas para detectar *God Class* utilizando el lenguaje específico de dominio.

```

1 RULE_CARD : Blob {
2   RULE : Blob { ASSOC : associated FROM : mainClass ONE TO : DataClass MANY } ;
3   RULE : MainClass { UNION LargeClassLowCohesion ControllerClass } ;
4   RULE : LargeClassLowCohesion { UNION LargeClass LowCohesion } ;
5   RULE : LargeClass { ( METRIC : NMD + NAD, VERY_HIGH , 0 ) } ;
6   RULE : LowCohesion { ( METRIC : LCOM5, VERY_HIGH , 20 ) } ;
7   RULE : ControllerClass { UNION
8     ( SEMANTIC : METHODNAME , {Process , Control , Ctrl , Command , Cmd,
9       Proc , UI , Manage , Drive })
10    ( SEMANTIC : CLASSNAME , {Process , Control , Ctrl , Command , Cmd,
11      Proc , UI , Manage , Drive , System ,
12      Subsystem} ) ;
13  } ;
14  RULE : DataClass { ( STRUCT : METHOD_ACCESSOR , 90 ) } ;
15 } ;

```

El método DECOR tiene una implementación como plugin de una herramienta propietaria Ptidej. No hemos utilizado esta herramienta, porque en nuestro contexto experimental sólo hemos considerado herramientas distribuidas como plugins de Eclipse.

4.3.3. Definición

En la Tabla 4.3 se recoge un conjunto parcial de entidades, resultado de aplicar el proceso definido en Sec. 4.1, el conjunto completo puede obtenerse en [Lóp12a]. Este conjunto de datos basado en entidades reales, es utilizado para definir el algoritmo de detección de *God Class* mediante aprendizaje, en la experimentación.

En la Tabla 4.3 sólo se muestran métricas de tamaño (LOC), de cohesión (LCOM) de complejidad (WMC) y de acoplamiento (RPC). Las entidades con defecto *God Class* (GC), se han localizado principalmente en entidades cuya naturaleza de diseño ha sido clasificada como desconocida. Se destaca la identificación de este defecto en una entidad de tipo test, otra de tipo utilidad y otra tipo entidad.

4.3.4. Experimentación

En las Tablas 4.4 y 4.5 se muestran los árboles de decisión con las reglas de detección, adaptados para cada uno de los contextos de detección estudiados. En las hojas del árbol, el primer valor encerrado entre paréntesis indica el número total de instancias (peso) que llegan a la hoja. El segundo, es el número de instancias mal clasificadas.

Tabla 4.3. Conjunto de datos parciales de *God Class*

Nombre de la Entidad	LOC	LCOM	WMC	RPC	...	Nat.	GC
org.jfree.chart.text.TextUtilities	761.0	0.83	78	68		util	true
org.jfree.data.general.- DatasetUtilities	2179.0	0	346	94		util	false
org.jfree.chart.axis.junit.- SegmentedTimelineTests	1092	0.93	81	92		test	true
org.jfree.chart.axis.junit.- ValueAxisTests	179	1	6	39		test	false
org.jfree.chart.plot.PiePlot	3460	0.97	341	247		unknown	true
org.jfree.chart.plot.PlotState	20	0	2	1		unknown	false
com.mountainminds.eclEmma.- internal.core.analysis.- JavaModelCoverage	116	0.833	20	24		core	true

Estos números serán decimales si en los datos existen valores perdidos de los atributos, el cálculo de este valor se discute en [Qui92, WFH11].

En ambas tablas se observa que cada aplicación tienen sus propias reglas de detección y que en ambos casos se incluye en las reglas el estereotipo de la entidad.

En la aplicación EclEmma 2.1.0, las reglas de detección incluyen métricas de complejidad (WMC) y de cohesión (LCOM), como mencionan otras heurísticas estudiadas (ver Sec. 4.3.1). Además se incluyen métricas relacionadas con la herencia (NOC y DIT), no consideradas en las definiciones iniciales. La heurística de dependencias con otras clases puede estar reflejada en la métrica (DIP), que sirve como indicador del principio de inversión de dependencias.

Tabla 4.4. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación EclEmma 2.1.0

Árbol J48 EclEmma 2.1.0	
i=con naturaleza j=con costes (5 veces más FN)	<pre> STEREOTYPE = unknown WMC <= 5: false (13.0) WMC > 5 NOC <= 0 DIP <= 0.286 DIT <= 1 DIP <= 0 LCOM <= 0.76: false (2.57/0.57) LCOM > 0.76: false (2.29) DIP > 0: false (3.64/1.43) DIT > 1: false (2.0) DIP > 0.286 WMC <= 8: false (2.5/0.5) WMC > 8: true (8.0/0.5) NOC > 0: false (2.0) STEREOTYPE = test: false (20.0) STEREOTYPE = interface: false (44.0) STEREOTYPE = util: false (9.0) STEREOTYPE = control: false (41.0) STEREOTYPE = core: true (2.0/1.0) </pre>

Tabla 4.5. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación JFreeChart 1.0.14

Árbol J48 JFreeChart 1.0.14	
i=con naturaleza j=con costes (5 veces más FN)	<pre> WMC <= 43: false (845.0) WMC > 43 WMC <= 101 LCOM (Podado 1 nivel) WMC > 101 stereotype = unknown NOTa <= 0: true (24.0) NOTa > 0 DIT <= 1: true (3.0/1.0) DIT > 1: true (3.0) stereotype = util: false (3.0/1.0) stereotype = control: true (0.0) stereotype = test: true (0.0) stereotype = core: true (0.0) stereotype = interface: true (0.0) stereotype = exception: true (0.0) </pre>

En la aplicación JFreeChart 1.0.14 las reglas de detección incluyen métricas de complejidad (WMC) y de cohesión (LCOM), como mencionan otras heurísticas estudiadas (ver Sec. 4.3.1). Por motivos de espacio y de mejora de comprensión, en la Tabla 4.5 no se presenta el nivel de las hojas del árbol correspondientes a la característica LCOM. La heurística de dependencia con otras clases, puede estar reflejada en la métrica de número de atributos de la clase (NOA).

De estas observaciones se puede resumir lo siguiente:

- Las reglas de detección que implementan las heurísticas de *God Class* utilizan los estereotipos de UML.
- Las reglas de detección que implementan las heurísticas de *God Class* tienen que ser adaptadas al contexto de aplicación.
- Aunque en las heurísticas de otros autores no existe relación entre *God Class* y herencia, en los casos estudiados sí que surge esta relación.

4.3.5. Validación

Las medidas para validar las reglas de identificación proporcionadas, se muestran en la Tabla 4.6.

Tabla 4.6. Evaluación de los clasificadores para el defecto *God Class*

	Precision	Recuperacion	F-Measure
JFreeChart 1.0.14	0.575	0.97	0.722
EclEmma 2.1.0	0.385	0.455	0.417

En el caso de JFreeChart, el valor de F-Measure está por encima del 0.66 recomendado

en la definición del proceso de gestión de defectos. Por tanto las reglas de detección pueden ser utilizadas por el auditor. Además el auditor puede supervisar la *tabla inicial de defectos-entidades*, generando una nueva *tabla de defectos-entidades revisada* (con FP y FN), para retroalimentar el proceso e intentar mejorar el valor de F-Measure.

En el caso de EclEmma, el valor de F-Measure está por debajo del 0.66 recomendado en la definición del proceso de gestión de defectos. Por tanto, se propone iterar sobre las diferentes tareas del proceso de gestión de defectos (definición, experimentación, validación) definido en la Sec. 4.1. El analista puede redefinir *God Class*, incluyendo más casos de ejemplo de los que disponga en su histórico, que manifiesten el defecto *God Class*.

4.4. Aplicación del proceso a Data Class

En esta sección se aplica el proceso de gestión de defectos de diseño, definido en la Sec.4.1, para el defecto *Data Class*. Se utilizan como productos de entrada (histórico de datos y códigos fuentes) los presentados en la Sec.4.2.1 y como criterio de aceptación el expuesto en la Sec. 4.2.2.

Las subsecciones se corresponden con los campos de la plantilla propuesta para presentar la aplicación del proceso para un defecto (ver Sec.4.2.3).

4.4.1. Clasificación y motivación

Este defecto se recoge en el catálogo de Fowler [FBB⁺99] y por Riel [Rie96]. Wake [Wak03] lo clasifica en la categoría que denomina de “entre clases” y en la subcategoría “de datos”.

Las clases de datos (Data Class) son copias de datos de las que dependen fuertemente otras clases. La carencia de funcionalidad definida en sus métodos, puede indicar que los datos y el comportamiento no están encapsulados en la misma clase. Esta característica rompe el principio de encapsulamiento, preservado en los buenos diseños orientados a objetos. Las clases de datos dificultan el mantenimiento, el desarrollo de pruebas y la comprensión del sistema.

Como excepciones documentadas a esta definición, están las aplicaciones que usan bibliotecas y frameworks de persistencia, como Hibernate. El código autogenerado para mapear los datos disponibles en sistemas de persistencia a objetos en memoria, genera clases de datos.

4.4.2. Heurísticas para su detección

Marinescu [LM06] propone basarse en tres características para definir su estrategia de detección:

1. La clase revela datos en lugar de ofrecer servicios. La mayoría de la interfaz de la clase expone datos en lugar de servicios. Se utiliza la métrica WOC (Weight Of Class) para modelar esta heurística.
2. La clase revela muchos atributos y no es compleja. Para esta heurística se consideran dos casos. El primero considera clases de datos no muy largas, sin funcionalidad (WMC) y sólo proporciona algunos datos (NOPA Number Of Public Attribute) y métodos de acceso (NOAM Number Of Accessor Methods). El segundo caso considera una clase grande, que aparentemente parece normal, excepto porque su gran interfaz pública proporciona algunos servicios y un número significativo de datos y métodos de acceso.

La estrategia expresada como conjunto de reglas es la siguiente:

```
DataClass := ((WOC < Percentil(25%)) and
              ((NOPA + NOAM) > FEW) and ((WMC < HIGH)) or
              ((NOPA + NOAM > MANY) and (WMC < VERY HIGH)))
```

Moha et.al [MGDL10] proponen que las clases de datos son aquellas con más del 90 % de métodos de acceso. La regla expresada con el lenguaje de dominio que proponen es la siguiente:

```
RULE : DataClass { ( STRUCT : METHOD_ACCESSOR , 90) } ;
```

4.4.3. Definición

En la Tabla 4.7 se recoge un conjunto parcial de entidades en la aplicación del proceso definido en Sec. 4.1, el conjunto completo puede obtenerse en [Lóp12a]. Este conjunto de datos basado en entidades reales, se utiliza para definir el algoritmo de detección de *Data Class*, mediante aprendizaje en la experimentación.

En la Tabla 4.7 sólo se muestran las métricas que han resultado relevantes en la clasificación: métricas de tamaño (LOC, EXEC Executables Sentences, NOA Number Of Attributes), de comentarios (DoC Density Of Comments) y de acoplamiento (RPC Response For Class). Las entidades con defecto *Data Class* (DC) se han localizado principalmente en entidades cuya naturaleza de diseño ha sido clasificada como desconocida. Se destaca la identificación de este defecto en una entidad de tipo control, dos de interfaz y otras dos de entidad.

4.4.4. Experimentación

En las Tablas 4.8 y 4.9 se muestran los árboles de decisión, con las reglas de detección, adaptados para cada uno de los contextos de detección estudiados.

Se observa que cada aplicación tienen sus propias reglas de detección y que, en ambos casos, no se incluye en las reglas el estereotipo de la entidad.

Tabla 4.7. Conjunto de datos parciales de Data Class

Nombre de la Entidad	LOC	EXEC	NOA	RPC	DoC	Nat.	DC
com.mountainminds.eclEmma.- internal.ui.UIMessages	142	0	121	2	0.007	interfaz	true
org.jfree.chart.- MouseWheelHandlers	95	11	2	18	0.37	control	true
org.jfree.chart.entity.- PieSectionEntity	183	10	5	7	0.50	entity	true
org.jfree.chart.labels.- ItemLabelPosition	139	4	5	4	0.4	unknown	true

En la aplicación EclEmma 2.1.0 las reglas de detección incluyen métricas de tamaño relacionada con el número de atributos (NOA), tal y como mencionan otras heurísticas estudiadas (ver Sec. 4.4.1). Además se incluyen otras métricas de tamaño (LOC) y referentes a la estructura como el número de clases internas e interfaces que implementa (NOT).

Tabla 4.8. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación EclEmma 2.1.0

Árbol J48 EclEmma 2.1.0	
i=con naturaleza j=con costes (5 veces más FN)	<pre> NOA <= 9: false (142.0) NOA > 9 NOT <= 1 LOC <= 79: false (2.0/1.0) LOC > 79: true (2.0) NOT > 1: false (6.0) </pre>

En la aplicación JFreeChart 1.0.14 las reglas de detección incluyen métricas de tamaño (NOA, EXEC), tal y como mencionan las heurísticas de otras propuestas estudiadas (ver Sec. 4.3.1). En el árbol no se considera la complejidad (WMC) considerada en otras propuestas. Sin embargo se utilizan nuevas medidas de acoplamiento (RFC, NOTe) y de documentación (DoC). Por motivos de espacio y de mejora de comprensión, en la Tabla 4.9 no se presentan las hojas del árbol correspondientes a la característica LCOM.

De estas observaciones se puede resumir lo siguiente:

- Las reglas de detección que implementan las heurísticas de *Data Class* tienen que ser adaptadas al contexto de aplicación.
- Aunque en las heurísticas estudiadas de otros autores existe relación entre *Data Class* y la complejidad medida con WMC, en los casos estudiados no surge esta relación.
- Incluir la medida de densidad de comentarios en la *Data Class* puede ayudar a identificarla.

Tabla 4.9. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación JFreeChart 1.0.14

	Árbol J48 JFreeChart 1.0.14
i=con naturaleza j=con costes (5 veces más FN)	<pre> NOA <= 1: false (574.07) NOA > 1 RFC <= 20 EXEC <= 2: false (46.5) EXEC > 2 RFC <= 2 LCOM (Podado 1 nivel) RFC > 2 NOA <= 3 EXEC <= 10: false (51.92) EXEC > 10 NOTE <= 0: true (4.53) NOTE > 0 EXEC <= 32: false (10.03) EXEC > 32: true (4.56/0.03) NOA > 3 DC <= 0.395349: false (20.57) DC > 0.395349 LCOM (Podado 1 nivel) RFC > 20: false (229.75) </pre>

4.4.5. Validación

Las medidas para validar las reglas de identificación proporcionadas, se muestran en la Tabla 4.10.

Tabla 4.10. Evaluación de los clasificadores para el defecto *Data Class*

	Precision	Recuperacion	F-Measure
JFreeChart 1.0.14	0.304	0.5	0.378
EclEmma 2.1.0	0.5	0.333	0.4

En ambos casos no se supera el criterio de validación aconsejado (F-Measure < 0.66). Se necesita redefinir el defecto *Data Class*, para ello se proponen dos alternativas complementarias:

- Añadir nuevas instancias de entidades con defecto *Data Class*, procedentes de un histórico de otras aplicaciones, de contexto similar.
- Actualizar los resultados obtenidos de la Tabla inicial de Defectos-Entidades (ver Tabla 4.7). En este sentido, hay que destacar que algunas de las entidades clasificadas inicialmente con el defecto *Data Class* realmente no lo tienen. Este es el caso de la clase `com.mountainminds.eclEmma.internal.ui.UIMessages`, generada automáticamente por Eclipse para enfrentarse al problema de la internacionalización. Otra clase que aparentemente está bien diseñada y mal clasificada es `org.jfree.chart.entity.PieSectionEntity`. Es interesante observar que esta clase puede ser estereotipada como «entity» y está documentada en [FBB⁺99] como una excepción a la definición.

4.5. Aplicación del proceso a Feature Envy

En esta sección se aplica el proceso de gestión de defectos de diseño definido en la Sec.4.1 para el defecto *Feature Envy*. Se utilizan como productos de entrada (histórico de datos y códigos fuentes) los presentados en la Sec.4.2.1 y como criterio de aceptación el expuesto en la Sec. 4.2.2.

Las subsecciones se corresponden con los campos de la plantilla propuesta para presentar la aplicación del proceso sobre un defecto (ver Sec.4.2.3).

4.5.1. Clasificación y motivación

Este defecto se recoge en el catálogo de Fowler [FBB⁺99] y se clasifica por Wake [Wak03] en la categoría que denomina “entre clases” y la subcategoría “de responsabilidad”.

Un método tiene el defecto *Feature Envy* si parece estar más preocupado en manipular datos de otras clases que de la suya propia. Está relacionado con la responsabilidad de la clase, ya que contiene métodos que deberían estar en otra clase.

Como excepciones Fowler en [FBB⁺99] indica que existen varios patrones de diseño que rompen esta regla, como el Visitante y el Estrategia.

4.5.2. Heurísticas para su detección

Marinescu [LM06] propone definir la estrategia de detección en base a contar el número de acceso a datos de otras clases, directamente o por métodos de acceso. *Feature Envy* sucede cuando los métodos proceden de muy pocas clases o sólo de una. La estrategia para detectar este defecto es la siguiente:

1. Métodos que utilizan sólo unos pocos atributos de otras clases. Se define la métrica de acceso de datos foráneos (ATFD Access To Foreign Data), utilizada también en *God Class*.
2. Métodos que usan más atributos de otras clases que de la clase donde están definidos. Se define la métrica LAA (Locality of Attribute Accesses).
3. Se accede a atributos foráneos que pertenecen a muy pocas clases. Se define la métrica FDP (Foreign Data Provider).

La estrategia expresada como conjunto de reglas es la siguiente:

```
FeatureEnvy := ((ATFD > FEW ) and
                (LAA < ONE THIRD) and
                (FDP <= FEW))
```

En la propuesta de Moha et.al [MGDL10] no se considera este defecto.

4.5.3. Definición

En la Tabla 4.11 se recoge un conjunto parcial de entidades, a las que se ha aplicado el proceso definido en Sec. 4.1, el conjunto completo puede obtenerse en [Lóp12a]. Este conjunto de datos basado en entidades reales, se utiliza para definir el algoritmo de detección de *Feature Envy* (FE) mediante aprendizaje en la experimentación.

En la Tabla 4.11 sólo se muestran métricas que han resultado relevantes en la clasificación: métricas de tamaño (NCLOC Non Comment Lines of Code, EXEC Executables Sentences), de acoplamiento (NP Number of Parameter) y de complejidad (V(G) Cyclomatic Complexity). Las entidades con defecto *Feature Envy* (FE) se han localizado, principalmente, en entidades cuya naturaleza de diseño ha sido clasificada como test y desconocida. Se destaca la identificación de este defecto en cuatro entidades de tipo interfaz.

Tabla 4.11. Conjunto de datos parciales de *Feature Envy*

Nombre de la Entidad	NCLOC	EXEC	NP	VG	...	Nat.	FE
org.jfree.chart.block- .junit.LabelBlockTests- testEquals()	40.0	30.0	0	1.0		test	true
org.jfree.chart.JFreeChart- drawTitle(Title, Graphics2D,- Rectangle2D, boolean)	64	14	4	10		inter.	true
org.jfree.chart.plot.MeterPlot- drawArcForInterval- (Graphics2D,Rectangle2D, MeterInterval)	16	4	3	4		unknown	true

4.5.4. Experimentación

En las Tablas 4.13 y 4.12, se muestran los árboles de decisión con las reglas de detección, adaptados para cada uno de los contexto de detección estudiados. En las hojas del árbol, el primer valor, encerrado entre paréntesis, indica el número total de instancias (peso) que llegan a la hoja. El segundo, es el número de instancias mal clasificadas. Estos números serán decimales si en los datos existen valores perdidos de los atributos. El cálculo de este valor se discute en [Qui92, WFH11].

Se observa que cada aplicación tienen sus propias reglas de detección y que, en ambos casos, se incluye en las reglas el estereotipo de la entidad.

En la aplicación JFreeChart 1.0.14, las reglas de detección incluyen las mismas métricas que en el caso de EclEmma 2.1.0, y además una de comentarios (DC). Además se observa la gran complejidad del árbol utilizado para clasificar.

Tabla 4.12: Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación JFreeChart 1.0.14

Árbol J48 JFreeChart 1.0.14	
i=con naturaleza j=con costes (5 veces más FN)	<pre> LOC <= 15 EXEC <= 2 NP <= 3: false (7840.59) NP > 3 NCLOC <= 5: false (167.76) NCLOC > 5 LOC <= 6 EXEC <= 1 EXEC <= 0: false (2.92) EXEC > 0: true (10.69/0.97) EXEC > 1: false (6.8) LOC > 6: false (64.15) EXEC > 2 EXEC <= 9 DC <= 0.105263 LOC <= 10 NCLOC <= 9 stereotype = unknown: false (477.4) stereotype = test NCLOC EXEC LOC podado 6 niveles stereotype = core: false (11.6) stereotype = control: false (4.86) stereotype = util: false (19.25) stereotype = interface: false (13.61) stereotype = exception: false (0.0) NCLOC > 9 EXEC <= 6 V(G) <= 2 EXEC <= 3 stereotype = unknown: true (5.83/0.97) stereotype = test: false (2.92) stereotype = core: false (0.0) stereotype = control: false (0.0) stereotype = util: false (0.97) stereotype = interface: false (0.0) stereotype = exception: false (0.0) EXEC > 3 EXEC <= 5: false (18.47) EXEC > 5: false (18.47/4.86) V(G) > 2: false (11.66) EXEC > 6: false (22.35) LOC > 10: false (510.26) DC > 0.105263 V(G) <= 1 NT <= 2 CLOC <= 1 stereotype = unknown: false (19.44) stereotype = test NCLOC <= 7 EXEC <= 3: false (2.92) EXEC > 3 EXEC <= 5 LOC <= 5: true (4.86) LOC > 5 LOC <= 6: true (5.83/0.97) LOC > 6: true (7.78/2.92) EXEC > 5: true (4.86) NCLOC > 7: false (3.89) stereotype = core: false (0.0) stereotype = control: false (0.97) stereotype = util: false (0.0) stereotype = interface: false (0.0) stereotype = exception: false (0.0) CLOC > 1: false (31.1) NT > 2: true (4.86) V(G) > 1: false (76.78) EXEC > 9 stereotype = unknown: false (27.12) stereotype = test EXEC LOC EXEC NLOC 6 niveles stereotype = core: false (0.12) stereotype = control: false (0.0) stereotype = util: false (2.97) stereotype = interface: false (6.8) stereotype = exception: false (0.0) LOC > 15 NCLOC <= 64 </pre>
Continúa en la página siguiente	

Árbol J48 JFreeChart 1.0.14	
<p>i=con naturaleza j=con costes (5 veces más FN)</p>	<pre> EXEC <= 13 NP <= 7 NP <= 1 DC NCLOC LOC NP EXEC VG podado 6 niveles<= 0.057971 NP > 1 stereotype = unknown CLOC, LOC, NCLOC, NP, VG podado 7 niveles stereotype = test CLOC <= 3 CLOC <= 0: true (5.83/0.97) CLOC > 0: false (3.89) CLOC > 3: true (4.86) stereotype = core: false (3.89) stereotype = control: false (4.86) stereotype = util NP <= 3: false (43.74) NP > 3 NP <= 4 V(G) <= 7: true (4.86) V(G) > 7: false (3.89) NP > 4: false (10.69) stereotype = interface: false (1.94) stereotype = exception CLOC <= 3: true (5.83/0.97) CLOC > 3: true (6.8/1.94) NP > 7 NT <= 1 NP <= 10 NP <= 8 NCLOC <= 12: true (11.66/1.94) NCLOC > 12: false (5.83) NP > 8: true (19.44) NP > 10 CLOC <= 5: false (8.75) CLOC > 5: true (5.83/0.97) NT > 1: false (2.92) EXEC > 13 EXEC <= 14 DC <= 0.045455 DC <= 0.028571 stereotype = unknown: false (8.75) stereotype = test NCLOC <= 19 LOC <= 17: true (4.86) LOC > 17 NCLOC <= 17: false (3.89) NCLOC > 17: true (5.83/0.97) NCLOC > 19: false (10.69) stereotype = core: false (0.0) stereotype = control: false (0.0) stereotype = util: false (0.97) stereotype = interface: true (4.86) stereotype = exception: false (0.0) DC > 0.028571 DC <= 0.038462 CLOC <= 1: true (38.88) CLOC > 1: true (10.69/0.97) DC > 0.038462 LOC <= 23: true (4.86) LOC > 23: false (2.92) DC > 0.045455: false (13.61) EXEC > 14 V(G) <= 8 CLOC <= 0 LOC <= 53 NCLOC <= 39 stereotype = unknown NCLOC <= 35: false (22.35) NCLOC > 35 NCLOC <= 36: true (9.72) NCLOC > 36: false (2.92) stereotype = test EXEC <= 15 NCLOC <= 20: false (2.92) NCLOC > 20 LOC <= 23: true (9.72) LOC > 23: true (7.78/2.92) </pre>
Continúa en la página siguiente	

Árbol J48 JFreeChart 1.0.14	
i=con naturaleza j=con costes (5 veces más FN)	<pre> EXEC > 15 V(G) <= 1: false (101.08/38.88) V(G) > 1: false (9.72) stereotype = core: false (0.0) stereotype = control: false (0.0) stereotype = util: false (0.97) stereotype = interface: false (9.72) stereotype = exception: false (0.0) NCLOC > 39 V(G) <= 5 NCLOC <= 43: true (17.49/2.92) NCLOC > 43: false (3.89) V(G) > 5: true (4.86) LOC > 53: false (19.44) CLOC > 0 LOC <= 37: false (84.56) LOC > 37 stereotype = unknown: false (17.49) stereotype = test NP, CLOC, LOC, VG EXEC NCLOC podado 8 niveles stereotype = core: false (0.0) stereotype = control: false (0.0) stereotype = util: false (0.0) stereotype = interface: false (0.0) stereotype = exception: false (0.0) V(G) > 8: false (35.96) NCLOC > 64: false (153.56) </pre>

En la aplicación EclEmma 2.1.0, las reglas de detección incluyen métricas de tamaño (NCLOC, EXEC), de acoplamiento (NP) y de complejidad (VG). Estas métricas no se han considerado en las heurísticas estudiadas previamente (ver Sec. 4.5.2).

Tabla 4.13. Árbol de decisión generado con algoritmo J48 con los entidades de la aplicación EclEmma 2.1.0

Árbol J48 EclEmma 2.1.0	
i=con naturaleza j=con costes (5 veces más FN)	<pre> Classifier Model J48 unpruned tree ----- NCLOC <= 14: false (736.54) NCLOC > 14 NP <= 0 V(G) <= 1: false (6.87) V(G) > 1 stereotype = unknown: false (0.98) stereotype = core: true (0.0) stereotype = control: true (0.0) stereotype = util: true (4.9) stereotype = test: true (0.0) stereotype = interface EXEC <= 1: true (4.9) EXEC > 1 CLOC <= 1: true (7.85/2.94) CLOC > 1: true (5.88/0.98) stereotype = exception: true (0.0) NP > 0: false (47.08) </pre>

De estas observaciones se puede resumir lo siguiente:

- Las reglas de detección que implementan las heurísticas de *Feature Envy* utilizan los estereotipos de UML.
- Las reglas de detección que implementan las heurísticas de *Feature Envy* tienen que ser adaptadas al contexto de aplicación.

- Las métricas de tamaño del método (EXEC, NLOC) y de acoplamiento (NP) pueden ayudar en la detección del defecto.

4.5.5. Validación

Las medidas para validar las reglas de identificación proporcionadas, se muestran en la Tabla 4.14.

Tabla 4.14. Evaluación de los clasificadores para el defecto *Feature Envy*

	Precision	Recuperacion	F-Measure
JFreeChart 1.0.14	0.256	0.375	0.304
EclEmma 2.1.0	0	0	0

En el caso de JFreeChart el valor de F-Measure está por debajo del 0.66, recomendado en la definición del proceso de gestión de defectos. Por tanto, se propone iterar sobre las diferentes tareas del proceso de gestión de defectos (definición, experimentación, validación) definido en la Sec. 4.1. El analista puede redefinir *Feature Envy* añadiendo, eliminando o modificando instancias. En este caso particular, después de analizar las instancias que conforman la definición del defecto, se observa que muchos de los métodos con este defecto (59 de 88) pertenecen al estereotipo «test». Nosotros consideramos que la naturaleza del diseño de este tipo de métodos es un caso excepcional, siendo estas instancias falsos positivos.

En el caso de EclEmma el valor de F-Measure está por debajo del 0.66, recomendado en la definición del proceso de gestión de defectos. Por tanto, se propone iterar sobre las diferentes tareas del proceso de gestión de defectos (definición, experimentación, validación) definido en la Sec. 4.1. En este caso el analista puede redefinir *Feature Envy*, incluyendo más casos de ejemplo de entidades con *Feature Envy* disponibles en su histórico.

4.6. Estudio empírico para evaluar la eficiencia en métodos que identifican defectos

En el proceso definido en la Sec.4.1, se propone generar nuevas heurísticas basadas en métricas de código, aplicando técnicas de aprendizaje supervisado, mediante clasificadores de tipo árbol de decisión. Nuestro trabajo se basa en la idea de que “el auditor”, encargado de identificar defectos en un sistema, es el conocedor del contexto del problema de la entidad y podría desear incluir información de diseño de la entidad (naturaleza de la entidad), para ser considerada en la tarea de identificación de defectos. Esta nueva funcionalidad, puede provocar un cambio en las heurísticas de detección para adaptarse a la nueva información de diseño. Por ello, es necesario realizar un estudio empírico para evaluar si la eficiencia de identificación de defectos mejora con las nuevas heurísticas.

La pregunta de investigación en este estudio empírico se puede enunciar como:

¿Influye la naturaleza de la entidad, modelada como estereotipo UML, en la predicción de defectos de diseño?

En las Sec. 4.7, 4.8 y 4.9 se van a presentar tres réplicas de un caso de estudio, para responder a la pregunta de investigación de manera empírica. Cada réplica consiste en aplicar el mismo proceso de experimentación sobre los siguientes defectos: *God Class*, *Data Class* y *Feature Envy*.

El desarrollo de los casos de estudio, según la clasificación dada por Robson [Rob02], ha seguido las guías para conducir la experimentación en ingeniería del software expuestas en [RH09] y el proceso experimental propuesto en [WRH⁺00], donde se especifican las siguientes fases: definición, planificación, operación, análisis, validez y presentación.

4.7. Evaluación de la eficiencia de métodos de identificación del defecto *God Class*

El objetivo principal de este estudio enunciado en formato GQM [BCR94] es el siguiente:

- *Analizar* entidades de código
- *Con el propósito* de estudiar el impacto de su clasificación, según su naturaleza de diseño basada en estereotipos UML
- *Con respecto* a la eficiencia en la identificación de defectos de diseño, en particular el defecto *God Class*
- *Desde el punto de vista* de los investigadores
- *en el contexto* académico de la Universidad de Burgos (UBU) y la Universidad de Valladolid (UVa) y de dos aplicaciones de código abierto JFreeChart 1.0.14 y EclEmma 2.1.0.

Como objetivo secundario del estudio se plantea:

- *Analizar* entidades de código
- *Con el propósito* de comparar dos métodos de clasificación (InCode y JDeodorant)
- *Con respecto* a la predicción del defecto de diseño *God Class*
- *Desde el punto de vista* de los investigadores
- *en el contexto* académico de la Universidad de Burgos (UBU) y la Universidad de Valladolid (UVa) y de dos aplicaciones de código abierto JFreeChart 1.0.14 y EclEmma 2.1.0.

4.7.1. Definición del caso de estudio: hipótesis y variables

Este estudio se ha realizado siguiendo las recomendaciones dadas en [WRH⁺00, KAKB⁺08, RH09].

Del objetivo principal del estudio, se deriva la siguiente hipótesis nula (H_0):

La naturaleza de la entidad de código, entendida como estereotipos estandar de UML, no mejora la eficiencia de los métodos de identificación de God Class, basados en métricas de código.

Las variables independientes son, por un lado, las medidas de la entidad de código. Por otro lado, la naturaleza de la entidad, que tiene una escala nominal y cuyas categorías se corresponden con los estereotipos estándar UML especificados en la Tabla 3.4.

Las variables dependientes son “la predicción binaria (true,false) indicando si la entidad de código tiene el defecto *God Class*” y la eficiencia de la clasificación medida con F-Measure.

Este tipo de problemas genera conjuntos de datos no balanceados, hay más entidades de código sin defecto que con defecto. En este escenario interesa penalizar la detección de falsos negativos. En [WFH11] se propone definir una matriz de costes como posible solución al balanceo de datos. Por ello, hemos considerado otro factor que puede afectar al resultado: el coste de equivocarse en una “no predicción” de una entidad con defecto (coste de falsos negativos). Este coste se mide con escala nominal con las categorías “sin coste”, “con coste”. La variable dependiente costes es tratada y justificada con mayor detalle en la Sec.4.7.2.1.

4.7.2. Planificación

En esta sección se presenta el diseño del caso de estudio (Sec. 4.7.2.1), información relativa a los sujetos y objetos del estudio (Sec. 4.7.2.2), y cuestiones relacionadas con la instrumentación para llevar a cabo el estudio (Sec. 4.7.2.3).

4.7.2.1. Diseño y variables de estudio

Las variables independientes del caso de estudio son:

- Medidas de código orientado a objeto, bien conocidas, de Chidamber y Kemerer [CK91], Lorenz y Kid [LK94] y otras. En la Tabla 4.15 se presentan las descripciones e identificadores de las medidas utilizadas.
- Naturaleza de la entidad, tiene una escala nominal, cuyas categorías se corresponden con estereotipos estándar UML especificados en la Tabla 3.4. La naturaleza de la entidad también se referencia en el texto como intención de diseño de la entidad.
- Coste de falsos negativos, se mide con escala nominal con las categorías “sin coste”, “con coste”

Las variables dependientes son:

Tabla 4.15. Conjunto parcial de métricas de código definido en RefactorIt

Descripción	Identificador
Density of Comments	DC
Executable Statements	EXEC
Total Lines of Code	LOC
Depth in Tree	DIT
Number of Children	NOC
Number of Fields	NOF
Response for Class	RFC
Weighted Methods per Class	WMC
Number of Attributes	NOA
Dependency Inversion Principle	DIP
Lack of Cohesion of Methods	LCOM

- Predicción binaria (true,false) indicando si la entidad de código tiene el defecto *God Class*.
- Eficiencia de la clasificación medida con F-Measure.

Se van a simular dos procesos de aprendizaje supervisado (con naturaleza y sin naturaleza) basados en la generación de clasificadores obtenidos mediante el algoritmo J48 [WFH11]. La clasificación inicial de la entidad en “con defecto” y “sin defecto”, se calcula como la unión de las dos clasificaciones obtenidas por sendos expertos. Se consideran expertos a los resultados automáticos obtenidos por JDeodorant e InCode, herramientas que identifican automáticamente el defecto de diseño *God Class*.

Los conjuntos de datos resultantes son desequilibrados, es decir existen muchas más entidades “sin defecto” que “con defecto”. En [CHCK12] se recomienda aplicar el algoritmo de clasificación J48 vs C45 para conjuntos de datos desequilibrados con las opciones: no poda, no colapsar y corrección de Laplace.

Medidas de la eficiencia de los clasificadores

Las medidas para evaluar la eficiencia de los clasificadores generados, “con naturaleza” y “sin naturaleza” de la entidad, han sido la Recuperación, Precisión y F-Measure descritas en [WFH11]. El cálculo de estas medidas ha sido expuesto en la Sec. 4.2.2, titulada validación de las decisiones del analista.

En el problema de identificación de defectos de código los dos tipos de error, falsos positivos (FP) y falsos negativos (FN), podrían tener diferentes costes, dependiendo de la organización y el contexto de uso. Es decir, en los escenarios de aplicación del proceso de identificación de defectos sobre un sistema, nos pueden aparecer dos situaciones extremas: se sospecha que muchas de las entidades son defectuosas o, por el contrario, se sospecha que la mayoría no lo son. En el primer caso el coste de un FP, es decir, el coste de identificar como defectuosa una entidad que no lo es, puede penalizarse más que el caso de detectar un FN, clasificar como no defectuosa una que sí que lo es. En

el segundo caso, el coste de identificar un FP, puede penalizarse menos que el caso de detectar un FN.

En general, en el problema que nosotros estudiamos se presenta la segunda situación, se cuenta con una gran cantidad de entidades donde pocas de ellas son defectuosas. Por ello definimos una matriz de costes, para tenerla en cuenta en el proceso de aprendizaje cuando se genera el clasificador. Esto provoca un cambio de reglas de detección en función de los costes. Los subconjuntos de instancias de entrenamiento seleccionados como clase real en la validación cruzada, no son aleatorios, sino que se crean en función de los costes.

En la Tabla 4.16 se muestra la matriz de costes utilizada en este caso de estudio. Se ha considerado ponderar los costes de equivocarse al predecir un falso negativo cinco veces más que el resto. Dicho de otra forma, la organización prefiere mayor recuperación (se decrementa FN) a costa de perder precisión (se incrementa FP). Hemos elegido 5, como peso del coste de los FN por ser el valor, entre los explorados, que proporcionaba una eficiencia mejor (F-Measure).

Tabla 4.16. Tabla de costes

		Clase predicha	
		sí	no
Clase real	sí	0 (TP)	5 (FN)
	no	1 (FP)	0 (TN)

Diseño del caso de estudio El diseño propuesto en la Tabla 4.17, se basa en evaluar la eficiencia de los clasificadores al cruzar las variables naturaleza (n) y coste (c) con sus posibles valores binarios:

- Naturaleza: n con naturaleza y \bar{n} sin naturaleza
- Costes: c con costes y \bar{c} sin costes

Una interpretación de los datos de la Tabla 4.17 con respecto a la hipótesis es: si $MedidasRendimiento_{\bar{n},\bar{c}}$ son iguales o mejores que el resto, entonces no es interesante considerar ni el coste, ni la naturaleza en la identificación de defectos.

Tabla 4.17. Diseño del caso de estudio

	Con costes	Sin costes
Con naturaleza	$MedidasRendimiento_{n,c}$	$MedidasRendimiento_{n,\bar{c}}$
Sin naturaleza	$MedidasRendimiento_{\bar{n},c}$	$MedidasRendimiento_{\bar{n},\bar{c}}$

Test estadísticos utilizados La hipótesis secundaria planteada en el trabajo es:

Hay diferencias en la clasificación hecha por dos jueces (InCode, JDeodorant)

La estudiaremos usando la prueba no paramétrica de McNemar que sirve para comparar las puntuaciones de dos jueces en sí/no sobre k objetos. [SC88]. Donde,

- Los objetos a clasificar son las entidades de código de la aplicación
- La variable dependiente es “la entidad tiene, o no tiene, el defecto *God Class*”

4.7.2.2. Selección de sujetos y objetos

El sujeto que participa en el estudio es el autor de esta tesis, cuyo perfil principal de conocimientos pertenece al área de mantenimiento del software.

Las bibliotecas Java que son los objetos de este estudio son: JFreeChart 1.0.14, EclEmma 2.1.0. Ambas han sido obtenidas a través del repositorio de software de código abierto *SourceForge* (<http://sourceforge.net/>).

JFreeChart es una biblioteca escrita en Java que facilita a los desarrolladores crear diagramas de calidad en sus aplicaciones. Soporta muchos formatos de salida: componentes Swing, ficheros de imágenes (PNG, JPEG) y ficheros con formato vectorial (PDF, EPS y SVG). Está distribuida bajo licencia GNU-LGPL. Desde el punto de vista de tamaño, la aplicación contiene 976 clases y 244.850 líneas de código.

EclEmma es un plugin de Eclipse escrito en Java para calcular la cobertura de pruebas, y está disponible bajo la licencia Eclipse Public License. Acelera el ciclo de desarrollo de test lanzando los test JUnit a la vez que analiza su cobertura. Mejora el análisis de cobertura, resumiendo los resultados de cobertura y colorea las sentencias del código fuente que no han sido probadas. Desde el punto de vista de tamaño, la aplicación contiene 153 clases y 10.561 líneas de código.

En el estudio, las entidades de código a medir son las clases. Es decir cuando hablamos de número de entidades hablamos de número de clases. La selección de objetos se ha basado en el conocimiento de ambas bibliotecas por el sujeto, y en la frecuente utilización por la comunidad de desarrolladores en Java.

4.7.2.3. Instrumentación

En el estudio hemos utilizado las herramientas siguientes:

- Una herramienta para calcular métricas de las entidades de código, RefactorIt [Aqr02]. En el contexto de predicción de defectos de diseño basados en métricas que se plantea en este trabajo se necesita una herramienta que calcule un amplio conjunto de métricas orientadas a objetos y permita la exportación de resultados para su análisis posterior. Bajo estas premisas, la herramienta seleccionada para obtener las medidas ha sido RefactorIt [Aqr02]. En la Tabla 4.15 se describe el conjunto de las métricas a emplear en el estudio, todas ellas proporcionadas por la herramienta.
- Una herramienta desarrollada por los propio autores [LMC10] que ayuda a la clasificación de entidades de código según su intención de diseño, expresada como estereotipo estándar de UML (*exception, interface, entity, control, test, utility*)

- Dos plugins de Eclipse, Incode 2.07 [Int08] y JDeodorant 4.0.12 [JDe11], que permiten obtener una predicción del defecto *God Class* sobre las entidades de código de los objetos seleccionados en este caso de estudio. Se han seleccionado, Incode 2.07 [Int08] y JDeodorant 4.0.12 [JDe11], por la buena documentación de sus métodos, reflejada en sus artículos de investigación publicados [FTSC11, TCC08, FTC07, MG10, MGV10]. En la Tabla 4.18 se muestra el conjunto de defectos de diseño que identifica cada herramienta.

Tabla 4.18. Defectos de diseño identificados por Incode 2.07 y JDeodorant 4.0.12

Ámbito	Nombre Defecto	InCode 2.07	JDeodorant 4.0.12
Clase	Dataclass	X	
	Godclass	X	X
Método	Feature Envy	X	X
	Long Method		X
	Type checking		X

- Una herramienta de minería de datos, Weka 3.7.5 [Wai07], que permite generar clasificadores con diferentes métodos de caja blanca (JRIP, J48) y calcula las medidas del proceso de aprendizaje supervisado (Recuperación, Precisión, F-Measure). La herramienta se ha elegido, por cumplir estos requisitos y por su documentación [WFH11].
- Una herramienta estadística, R 2.7.1 [OS09], para realizar el test de hipótesis de McNemar.

4.7.3. Operación

En este apartado se presenta cómo se ha llevado a cabo el estudio.

La instrumentación ha sido realizada en un único ordenador personal en el mes de febrero de 2012. La descripción del entorno tecnológico es la siguiente:

- Intel(R) Core(TM) 2Quad CPU Q8300 2.5 GHz
- RAM 4GB
- Sistema Operativo Windows 7 Profesional
- Eclipse 3.7.1
- Java 1.7.0

La principal incidencia encontrada ha sido un problema de eficiencia de memoria cuando se trabaja con el plugin de Eclipse JDeodorant, especialmente relevante en el caso de analizar proyectos grandes. Para superar el problema se optó por ampliar la memoria de ejecución de la máquina virtual de Java que lanzaba Eclipse (-Xmx1024m), o realizar la recogida por cada módulo de la aplicación.

4.7.4. Análisis

Las preguntas que responderemos se derivan del objetivo de este estudio, propuesto en la Sec. 4.7. En función de ellas se plantean las correspondientes hipótesis, como se muestra en la Tabla 4.19.

Debido a la incorporación de la variable costes, la hipótesis principal $H_{0,1}$ se especifica a través de dos hipótesis $H_{0,1a}$ y $H_{0,1b}$.

Tabla 4.19. Preguntas e hipótesis

<p>$H_{0,1}$: ¿Influye la naturaleza de la entidad, modelada como estereotipo UML, en la predicción de defectos de diseño tipo <i>God Class</i>?</p>
<p>$H_{0,1a}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de <i>God Class</i> en el caso de estudio r. Donde, $r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$, $i \in \{ \text{sin naturaleza, con naturaleza} \}$</p> <p>$H_{0,1b}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de <i>God Class</i> en el caso de estudio r cuando el coste de falsos negativos es k. Donde, $r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$, $i \in \{ \text{sin naturaleza, con naturaleza} \}$, $k \in \{ \text{sin coste, con coste FN según Tabla 4.16} \}$</p>
<p>$H_{0,2}$: ¿Son similares los métodos respecto a la clasificación binaria <i>God Class</i>?</p>
<p>$H_{0,2}$ Los métodos de identificación (j), no se diferencian en cuanto a los resultados de identificación de <i>God Class</i> en el caso de estudio r. Donde, $r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$, $j \in \{ \text{Incode 2.07, JDeodorant 4.0.12} \}$</p>

4.7.4.1. Estudio de la eficiencia de los clasificadores para identificar God Class

En este apartado se estudia la hipótesis $H_{0,1}$, utilizando como medidas de la eficiencia de los clasificadores generados en el proceso de aprendizaje, la recuperación, la precisión y F-Measure.

En las Tablas 4.20 y 4.21 se presentan los resultados obtenidos al medir la eficiencia de los clasificadores generados en el proceso de aprendizaje. La interpretación de las tablas ha sido descrita en la Sec. 4.7.2.1. Se ha resaltado con sombreado gris la fila que contiene las medidas, utilizadas para evaluar la eficiencia de los clasificadores en el proceso de aprendizaje. Además hemos considerado necesario mostrar el resto de información, para describir el proceso de aprendizaje completo en los casos de estudio.

Tabla 4.20. Evaluación de los clasificadores para el defecto *God Class* sobre el conjunto de datos generado a partir de JFreeChart 1.0.14

		JFreeChart 1.0.14					
i=sin naturaleza j=sin costes	Correctly Classified Instances	918				94.1538 %	
	Incorrectly Classified Instances	57				5.8462 %	
	Total Number of Instances	975					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.971	0.463	0.966	0.971	0.969	false
		0.537	0.029	0.581	0.537	0.558	true
	Weighted Avg.	0.942	0.433	0.94	0.942	0.94	
	=== Confusion Matrix ===						
		a	b	<-- classified as			
	882	26	a = false				
	31	36	b = true				
i=sin naturaleza j=con costes	Correctly Classified Instances	924				94.7692 %	
	Incorrectly Classified Instances	51				5.2308 %	
	Total Number of Instances	975					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.946	0.03	0.998	0.946	0.971	false
		0.97	0.054	0.57	0.97	0.718	true
	Weighted Avg.	0.948	0.032	0.968	0.948	0.954	
	=== Confusion Matrix ===						
		a	b	<-- classified as			
	859	49	a = false				
	2	65	b = true				
i=con naturaleza j=sin costes	Correctly Classified Instances	920				94.359 %	
	Incorrectly Classified Instances	55				5.641 %	
	Total Number of Instances	975					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.974	0.463	0.966	0.974	0.97	false
		0.537	0.026	0.6	0.537	0.567	true
	Weighted Avg.	0.944	0.433	0.941	0.944	0.942	
	=== Confusion Matrix ===						
		a	b	<-- classified as			
	884	24	a = false				
	31	36	b = true				
i=con naturaleza j=con costes	Correctly Classified Instances	925				94.8718 %	
	Incorrectly Classified Instances	50				5.1282 %	
	Total Number of Instances	975					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.947	0.03	0.998	0.947	0.972	false
		0.97	0.053	0.575	0.97	0.722	true
	Weighted Avg.	0.949	0.031	0.969	0.949	0.955	
	=== Confusion Matrix ===						
		a	b	<-- classified as			
	860	48	a = false				
	2	65	b = true				

$H_{0,1a}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de *God Class* en el caso de estudio r.

Donde,

$r \in \{ \text{JFreeChart 1.0.14, EcEmma 2.1.0.} \}$,

Tabla 4.21. Evaluación de los clasificadores para el defecto *God Class* sobre el conjunto de datos generado a partir de EclEmma 2.1.0

	EclEmma 2.1.0						
i=sin naturaleza j=sin costes	Correctly Classified Instances	135					88.8158 %
	Incorrectly Classified Instances	17					11.1842 %
	Total Number of Instances	152					
	=== Detailed Accuracy By Class ===						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	
	0.943	0.818	0.937	0.943	0.94	false	
	0.182	0.057	0.2	0.182	0.19	true	
	Weighted Avg.	0.888	0.763	0.883	0.888	0.886	
	=== Confusion Matrix ===						
	a	b	<-- classified as				
	133	8	a = false				
	9	2	b = true				
i=sin naturaleza j=con costes	Correctly Classified Instances	133					87.5 %
	Incorrectly Classified Instances	19					12.5 %
	Total Number of Instances	152					
	=== Detailed Accuracy By Class ===						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	
	0.915	0.636	0.949	0.915	0.931	false	
	0.364	0.085	0.25	0.364	0.296	true	
	Weighted Avg.	0.875	0.596	0.898	0.875	0.885	
	=== Confusion Matrix ===						
	a	b	<-- classified as				
	129	12	a = false				
	7	4	b = true				
i=con naturaleza j=sin costes	Correctly Classified Instances	140					92.1053 %
	Incorrectly Classified Instances	12					7.8947 %
	Total Number of Instances	152					
	=== Detailed Accuracy By Class ===						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	
	0.957	0.545	0.957	0.957	0.957	false	
	0.455	0.043	0.455	0.455	0.455	true	
	Weighted Avg.	0.921	0.509	0.921	0.921	0.921	
	=== Confusion Matrix ===						
	a	b	<-- classified as				
	135	6	a = false				
	6	5	b = true				
i=con naturaleza j=con costes	Correctly Classified Instances	138					90.7895 %
	Incorrectly Classified Instances	14					9.2105 %
	Total Number of Instances	152					
	=== Detailed Accuracy By Class ===						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	
	0.943	0.545	0.957	0.943	0.95	false	
	0.455	0.057	0.385	0.455	0.417	true	
	Weighted Avg.	0.908	0.51	0.915	0.908	0.911	
	=== Confusion Matrix ===						
	a	b	<-- classified as				
	133	8	a = false				
	6	5	b = true				

$$i \in \{ \text{sin naturaleza, con naturaleza} \}$$

La estructura del análisis para la hipótesis $H_{0,1a}$ es la siguiente:

1. Analizar cómo influye en la eficiencia de la identificación del defecto *God Class* utilizar o no la naturaleza de la entidad, en el caso de estudio JFreeChart 1.0.14.
2. Analizar cómo influye en la eficiencia de la identificación del defecto *God Class* al utilizar o no la naturaleza de la entidad, en el caso de estudio EclEmma 2.1.0.
3. Comparar los resultados anteriores.

1. En el caso de estudio JFreeChart, las medidas de eficiencia, con respecto a la clase True (entidades con el defecto *God Class*), mejoran levemente cuando consideramos la naturaleza de la entidad, excepto la recuperación que se mantiene igual. Los datos que justifican la afirmación son los siguientes:

Sin naturaleza:	Precisión (0,581)	Recuperación (0,537)	F-Measure (0,558)
Con naturaleza:	Precisión (0,600)	Recuperación (0,537)	F-Measure (0,567)

2. En el caso de estudio EclEmma, las medidas de eficiencia, con respecto a la clase True (entidades con el defecto *God Class*), mejoran significativamente cuando consideramos la naturaleza de la entidad.

Sin naturaleza:	Precisión (0,200)	Recuperación (0,182)	F-Measure (0,190)
Con naturaleza:	Precisión (0,455)	Recuperación (0,455)	F-Measure (0,455)

3. En conclusión, utilizar la naturaleza de la entidad ha mejorado las medidas de eficiencia de detección de defectos *God Class*.

$H_{0,1b}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de *God Class* en el caso de estudio r cuando el coste de falsos negativos es k.

Donde,

$r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$,

$i \in \{ \text{sin naturaleza, con naturaleza} \}$,

$k \in \{ \text{sin coste, con coste FN según Tabla 4.16} \}$

La estructura del análisis para la hipótesis $H_{0,1b}$ es la siguiente:

1. Analizar cómo influye en la eficiencia de la identificación de defecto *God Class* utilizando o no la matriz de costes, al utilizar o no la naturaleza de la entidad, en el caso de estudio JFreeChart 1.0.14.
2. Analizar cómo influye en la eficiencia de la identificación de defecto *God Class* utilizando o no la matriz de costes, al utilizar o no la naturaleza de la entidad, en el caso de estudio EclEmma 2.1.0.

3. Comparar los resultados anteriores.

1. En el caso de estudio JFreeChart, las medidas de eficiencia cuando utilizamos la matriz de costes, con respecto a la clase True (entidades con el defecto *God Class*), mejoran al considerar la naturaleza de la entidad.

Sin naturaleza:	Precisión (0,570)	Recuperación (0,970)	F-Measure (0,718)
Con naturaleza:	Precisión (0,575)	Recuperación (0,970)	F-Measure (0,722)

Las medidas de eficiencia cuando utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *God Class*), mejoran al considerar la matriz de costes para la recuperación y F-Measure, cuyo aumento compensa la disminución de la precisión.

Sin matriz de costes:	Precisión (0,600)	Recuperación (0,537)	F-Measure (0,567)
Con matriz de costes:	Precisión (0,575)	Recuperación (0,970)	F-Measure (0,722)

Las medidas de eficiencia cuando no utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *God Class*), mejoran al considerar la matriz de costes para la recuperación y F-Measure cuyo aumento compensa la disminución de la precisión.

Sin matriz de costes: Precisión (0,581) Recuperación (0,537) F-Measure (0,558)
 Con matriz de costes: Precisión (0,570) Recuperación (0,970) F-Measure (0,718)

2. En el caso de estudio EclEmma, las medidas de eficiencia cuando utilizamos la matriz de costes, con respecto a la clase True (entidades con el defecto *God Class*), mejoran significativamente cuando consideramos la naturaleza de la entidad.

Sin naturaleza: Precisión (0,250) Recuperación (0,364) F-Measure (0,296)
 Con naturaleza: Precisión (0,385) Recuperación (0,455) F-Measure (0,417)

Las medidas de eficiencia cuando utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *God Class*), no mejoran, la F-Measure empeora ligeramente.

Sin matriz de costes: Precisión (0,455) Recuperación (0,455) F-Measure (0,455)
 Con matriz de costes: Precisión (0,385) Recuperación (0,455) F-Measure (0,417)

Las medidas de eficiencia cuando no utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *God Class*), mejoran al considerar la matriz de costes.

Sin matriz de costes: Precisión (0,200) Recuperación (0,182) F-Measure (0,190)
 Con matriz de costes: Precisión (0,250) Recuperación (0,364) F-Measure (0,296)

3. En conclusión, en el caso de estudio EclEmma hay un resultado no coincidente con el resto de los resultados cuando consideramos la naturaleza de la entidad. Sin embargo, teniendo en cuenta que la cantidad de entidades con defectos es muy pequeña, se pone claramente de relieve que la matriz de costes es necesaria: es significativo que cuando utilizamos la matriz de costes, utilizar la naturaleza de la entidad ha mejorado las medidas de la eficiencia para detectar el defecto *God Class* en ambos casos de estudio. Por ello, tanto la matriz de costes como la naturaleza de la entidad son dos factores relevantes para mejorar la detección de dicho defecto.

La Tabla 4.22 resume los resultados del análisis: Con o Sin Naturaleza (n , \bar{n}), Con o Sin Costes (c , \bar{c}).

Tabla 4.22. Resumen del análisis del caso de estudio *God Class* respecto $H_{0,1a}$ y $H_{0,1b}$

	Precisión	Recuperación	F-Measure
$\bar{c}: n$ vs. \bar{n}	Mejora en ambos casos	Mejora en ambos casos	Mejora en ambos casos
$\bar{n}: c$ vs \bar{c}	EclEmma mejora	Mejora en ambos casos	Mejora en ambos casos
$n: c$ vs \bar{c}	JFreeChart mejora	JFreeChart mejora	JFreeChart mejora
$c: n$ vs. \bar{n}	Mejora en ambos casos	Mejora en ambos casos	Mejora en ambos casos

4.7.4.2. Comparación de los métodos de clasificación

En este apartado se estudia la hipótesis $H_{0,2}$ utilizando el test estadístico de McNemar.

$H_{0,2}$ Los métodos de identificación (j), no se diferencian en cuanto a los resultados de identificación de *God Class* en el caso de estudio r .

Donde,

$r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$,

$j \in \{ \text{Incode 2.07, JDeodorant 4.0.12} \}$

La Tabla 4.23 contiene la información sobre la identificación de defectos de los casos de estudio con los dos métodos utilizados.

Tabla 4.23. Predicciones del defecto *God Class* con *JDeodorant* e *Incode*

	# total de entidades	# entidades con defecto en InCode	# entidades con defecto JDeodorant
JFreeChart	975	67	0
EclEmma	152	0	6

En la Tabla 4.24 se muestra el resumen estadístico utilizado para contrastar la hipótesis nula $H_{0,2}$. A partir del test de hipótesis de McNemar se rechaza la hipótesis nula ($\alpha \leq 0,05$) en los dos casos de estudio considerados. Así que podemos concluir que los métodos de *Incode* y *JDeodorant* no son similares. Estos resultados justifican la necesidad de adaptar el método de identificación por medio de aprendizaje supervisado.

Tabla 4.24. Análisis de la concordancia entre los métodos *Incode* y *JDeodorant*

	JFreeChart	EclEmma
p-valor	7.433e-16	0.04123
McNemar	65.0149	4.1667

4.7.4.3. Amenazas a la validez

La validez de construcción del caso de estudio, entendida como las variables que han sido correctamente medidas, está afectada por la variable naturaleza de la entidad, pues la clasificación de entidades en estereotipos UML no es ortogonal cuando se aplica sobre entidades de código reales.

La validez interna del caso de estudio, entendida como la causalidad de nuestras conclusiones, está afectada porque la clasificación de entidades en estereotipos UML es subjetiva. Aunque esta amenaza se reduce teniendo en cuenta los resultados analizados en la Sec. 3.6.5 y 3.7.

Desde la perspectiva de la validez externa, referida a cuántos de nuestros resultados se pueden generalizar en otras circunstancias, se han identificado las siguientes amenazas:

- Los proyectos utilizados y el lenguaje de programación de los mismos, limita la extensión de estos resultados.
- La selección de los sujetos se ha hecho por conveniencia.
- Los resultados dependen del algoritmo utilizado y su configuración para clasificar, en nuestro caso J48 -O -U -A.

4.8. Evaluación de la eficiencia de métodos de identificación del defecto Data Class

En esta sección se presenta una réplica del caso de estudio de la Sec. 4.7. La réplica se construye con el objetivo de minimizar la amenaza, referida a cuántos de nuestros resultados se pueden generalizar con otro tipo de defectos. En la réplica las tareas de definición, planificación, operación son las mismas que el caso de estudio original, cambiando la variable binaria dependiente *God Class* por *Data Class*. Por esto, en este apartado, sólo se presenta el análisis de los resultados.

El objetivo principal de este estudio, enunciado en formato GQM [BCR94] es el siguiente:

- *Analizar* entidades de código
- *Con el propósito de* estudiar el impacto de su clasificación, según su naturaleza de diseño basada en estereotipos UML
- *Con respecto a* la eficiencia en la identificación de defectos de diseño, en particular el defecto *Data Class*
- *Desde el punto de vista* de los investigadores
- *en el contexto* académico de la Universidad de Burgos (UBU) y la Universidad de Valladolid (UVa) y de dos aplicaciones de código abierto JFreeChart 1.0.14 y EclEmma 2.1.0.

Desde el punto de vista de instrumentación este defecto sólo es identificado por la herramienta InCode, por tanto se ha eliminado el objetivo secundario del estudio original. Además se añade en el propósito el impacto de tener en cuenta el coste, tal cómo se analizó en Sec. 4.7.1.

4.8.1. Análisis

Las preguntas que responderemos se derivan del objetivo de este estudio, propuesto en la Sec. 4.8. En función de ellas se plantean las correspondientes hipótesis, como se muestra en la Tabla 4.25.

Tabla 4.25. Preguntas e hipótesis

$H_{0,1}$: ¿Influye la naturaleza de la entidad, modelada como estereotipo UML, en la predicción de defectos de diseño tipo <i>Data Class</i> ?
$H_{0,1a}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de <i>Data Class</i> en el caso de estudio r. Donde, $r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$, $i \in \{ \text{sin naturaleza, con naturaleza} \}$
$H_{0,1b}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de <i>Data Class</i> en el caso de estudio r cuando el coste de falsos negativos es k. Donde, $r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$, $i \in \{ \text{sin naturaleza, con naturaleza} \}$, $k \in \{ \text{sin coste, con coste FN según Tabla 4.16} \}$

4.8.1.1. Estudio de la eficiencia de los clasificadores para identificar *Data Class*

En este apartado se estudia la hipótesis $H_{0,1}$, utilizando como medidas de eficiencia de los clasificadores generados en el proceso de aprendizaje, la recuperación, la precisión y F-Measure.

En las Tablas 4.26 y 4.27 se presentan los resultados obtenidos al medir la eficiencia de los clasificadores generados en el proceso de aprendizaje. La interpretación de las tablas ha sido descrita en la Sec. 4.7.2.1. Se ha resaltado con sombreado gris la fila que contiene las medidas derivadas utilizadas para evaluar la eficiencia de los clasificadores en el proceso de aprendizaje. Además, hemos considerado necesario mostrar el resto de información, para describir el proceso de aprendizaje completo en los casos de estudio.

$H_{0,1a}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de *Data Class* en el caso de estudio r.

Donde,
 $r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$,
 $i \in \{ \text{sin naturaleza, con naturaleza} \}$

La estructura del análisis para la hipótesis $H_{0,1a}$ es la siguiente:

1. Analizar cómo influye en la eficiencia de la identificación del defecto *Data Class* utilizando o no la naturaleza de la entidad, en el caso de estudio JFreeChart 1.0.14.

Tabla 4.26. Evaluación de los clasificadores para el defecto *Data Class* sobre el conjunto de datos generado a partir de JFreeChart 1.0.14

	JFreeChart 1.0.14						
i=sin naturaleza j=sin costes	Correctly Classified Instances	1062			97.3419 %		
	Incorrectly Classified Instances	29			2.6581 %		
	Total Number of Instances	1091					
	=== Detailed Accuracy By Class ===						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	
	0.992	0.75	0.98	0.992	0.986	false	
	0.25	0.008	0.467	0.25	0.326	true	
	Weighted Avg.	0.973	0.731	0.967	0.973	0.969	
	=== Confusion Matrix ===						
	a	b	<-- classified as				
	1055	8	a = false				
	21	7	b = true				
i=sin naturaleza j=con costes	Correctly Classified Instances	1045			95.7837 %		
	Incorrectly Classified Instances	46			4.2163 %		
	Total Number of Instances	1091					
	=== Detailed Accuracy By Class ===						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	
	0.97	0.5	0.987	0.97	0.978	false	
	0.5	0.03	0.304	0.5	0.378	true	
	Weighted Avg.	0.958	0.488	0.969	0.958	0.963	
	=== Confusion Matrix ===						
	a	b	<-- classified as				
	1031	32	a = false				
	14	14	b = true				
i=con naturaleza j=sin costes	Correctly Classified Instances	1063			97.4335 %		
	Incorrectly Classified Instances	28			2.5665 %		
	Total Number of Instances	1091					
	=== Detailed Accuracy By Class ===						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	
	0.992	0.714	0.981	0.992	0.987	false	
	0.286	0.008	0.5	0.286	0.364	true	
	Weighted Avg.	0.974	0.696	0.969	0.974	0.971	
	=== Confusion Matrix ===						
	a	b	<-- classified as				
	1055	8	a = false				
	20	8	b = true				
i=con naturaleza j=con costes	Correctly Classified Instances	1045			95.7837 %		
	Incorrectly Classified Instances	46			4.2163 %		
	Total Number of Instances	1091					
	=== Detailed Accuracy By Class ===						
	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	
	0.97	0.5	0.987	0.97	0.978	false	
	0.5	0.03	0.304	0.5	0.378	true	
	Weighted Avg.	0.958	0.488	0.969	0.958	0.963	
	=== Confusion Matrix ===						
	a	b	<-- classified as				
	1031	32	a = false				
	14	14	b = true				

2. Analizar cómo influye en la eficiencia de la identificación del defecto *Data Class* utilizando o no la naturaleza de la entidad, en el caso de estudio EclEmma 2.1.0.
3. Comparar los resultados anteriores.
1. En el caso de estudio JFreeChart, las medidas de eficiencia recuperación y F-Measure, con respecto a la clase True (entidades con el defecto *DataClass*), mejoran levemente cuando consideramos la naturaleza de la entidad, mientras que la precisión empeora. Los datos que justifican la afirmación son los siguientes:

Sin naturaleza: Precisión (0,467) Recuperación (0,25) F-Measure (0,326)
 Con naturaleza: Precisión (0,304) Recuperación (0,5) F-Measure (0,378)

Tabla 4.27. Evaluación de los clasificadores para el defecto *Data Class* sobre el conjunto de datos generado a partir de EclEmma 2.1.0

EclEmma 2.1.0																			
i=sin naturaleza j=sin costes	Correctly Classified Instances 151 99.3421 % Incorrectly Classified Instances 1 0.6579 % Total Number of Instances 152 Ignored Class Unknown Instances 51 === Detailed Accuracy By Class === <table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>TP Rate</th> <th>FP Rate</th> <th>Precision</th> <th>Recall</th> <th>F-Measure</th> <th>Class</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0.333</td> <td>0.993</td> <td>1</td> <td>0.997</td> <td>false</td> </tr> <tr style="background-color: #e6f2ff;"> <td>0.667</td> <td>0</td> <td>1</td> <td>0.667</td> <td>0.8</td> <td>true</td> </tr> </tbody> </table> Weighted Avg. 0.993 0.327 0.993 0.993 0.993 === Confusion Matrix === a b <-- classified as 149 0 a = false 1 2 b = true	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	1	0.333	0.993	1	0.997	false	0.667	0	1	0.667	0.8	true
TP Rate	FP Rate	Precision	Recall	F-Measure	Class														
1	0.333	0.993	1	0.997	false														
0.667	0	1	0.667	0.8	true														
i=sin naturaleza j=con costes	Correctly Classified Instances 148 97.3684 % Incorrectly Classified Instances 4 2.6316 % Total Number of Instances 152 Ignored Class Unknown Instances 51 <table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>TP Rate</th> <th>FP Rate</th> <th>Precision</th> <th>Recall</th> <th>F-Measure</th> <th>Class</th> </tr> </thead> <tbody> <tr> <td>0.98</td> <td>0.333</td> <td>0.993</td> <td>0.98</td> <td>0.986</td> <td>false</td> </tr> <tr style="background-color: #e6f2ff;"> <td>0.667</td> <td>0.02</td> <td>0.4</td> <td>0.667</td> <td>0.5</td> <td>true</td> </tr> </tbody> </table> Weighted Avg. 0.974 0.327 0.981 0.974 0.977 === Confusion Matrix === a b <-- classified as 146 3 a = false 1 2 b = true	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	0.98	0.333	0.993	0.98	0.986	false	0.667	0.02	0.4	0.667	0.5	true
TP Rate	FP Rate	Precision	Recall	F-Measure	Class														
0.98	0.333	0.993	0.98	0.986	false														
0.667	0.02	0.4	0.667	0.5	true														
i=con naturaleza j=sin costes	Correctly Classified Instances 150 98.6842 % Incorrectly Classified Instances 2 1.3158 % Total Number of Instances 152 Ignored Class Unknown Instances 51 === Detailed Accuracy By Class === <table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>TP Rate</th> <th>FP Rate</th> <th>Precision</th> <th>Recall</th> <th>F-Measure</th> <th>Class</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>0.667</td> <td>0.987</td> <td>1</td> <td>0.993</td> <td>false</td> </tr> <tr style="background-color: #e6f2ff;"> <td>0.333</td> <td>0</td> <td>1</td> <td>0.333</td> <td>0.5</td> <td>true</td> </tr> </tbody> </table> Weighted Avg. 0.987 0.654 0.987 0.987 0.984 === Confusion Matrix === a b <-- classified as 149 0 a = false 2 1 b = true	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	1	0.667	0.987	1	0.993	false	0.333	0	1	0.333	0.5	true
TP Rate	FP Rate	Precision	Recall	F-Measure	Class														
1	0.667	0.987	1	0.993	false														
0.333	0	1	0.333	0.5	true														
i=con naturaleza j=con costes	Correctly Classified Instances 149 98.0263 % Incorrectly Classified Instances 3 1.9737 % Total Number of Instances 152 Ignored Class Unknown Instances 51 === Detailed Accuracy By Class === <table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>TP Rate</th> <th>FP Rate</th> <th>Precision</th> <th>Recall</th> <th>F-Measure</th> <th>Class</th> </tr> </thead> <tbody> <tr> <td>0.993</td> <td>0.667</td> <td>0.987</td> <td>0.993</td> <td>0.99</td> <td>false</td> </tr> <tr style="background-color: #e6f2ff;"> <td>0.333</td> <td>0.007</td> <td>0.5</td> <td>0.333</td> <td>0.4</td> <td>true</td> </tr> </tbody> </table> Weighted Avg. 0.98 0.654 0.977 0.98 0.978 === Confusion Matrix === a b <-- classified as 148 1 a = false 2 1 b = true	TP Rate	FP Rate	Precision	Recall	F-Measure	Class	0.993	0.667	0.987	0.993	0.99	false	0.333	0.007	0.5	0.333	0.4	true
TP Rate	FP Rate	Precision	Recall	F-Measure	Class														
0.993	0.667	0.987	0.993	0.99	false														
0.333	0.007	0.5	0.333	0.4	true														

2. En el caso de estudio EclEmma, las medidas de eficiencia, con respecto a la clase True (entidades con el defecto *Data Class*), empeoran cuando consideramos la naturaleza de la entidad. Una posible razón de este efecto es el escaso número de entidades con el defecto *Data Class* (3 de 152).

Sin naturaleza: Precisión (1) Recuperación (0,67) F-Measure (0,8)
 Con naturaleza: Precisión (1) Recuperación (0,34) F-Measure (0,5)

3. En conclusión, comparando los resultados anteriores sólo se aprecia mejora en

el caso JFreeChart. A partir de las observaciones no se puede afirmar que la naturaleza de la entidad ha mejorado las medidas de eficiencia de detección de defectos del tipo *Data Class*.

$H_{0,1b}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de *Data Class* en el caso de estudio r cuando el coste de falsos negativos es k.

Donde,

$r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$,

$i \in \{ \text{sin naturaleza, con naturaleza} \}$,

$k \in \{ \text{sin coste, con coste FN según Tabla 4.16} \}$

La estructura del análisis para la hipótesis $H_{0,1b}$ es la siguiente:

1. Analizar cómo influye en la eficiencia de la identificación de defecto *Data Class* utilizando o no la matriz de costes, al utilizar o no la naturaleza de la entidad, en el caso de estudio JFreeChart 1.0.14.
 2. Analizar cómo influye en la eficiencia de la identificación de defecto *Data Class* utilizando o no la matriz de costes, al utilizar o no la naturaleza de la entidad, en el caso de estudio EclEmma 2.1.0.
 3. Comparar los resultados anteriores.
1. En el caso de estudio JFreeChart, las medidas de eficiencia cuando utilizamos la matriz de costes, con respecto a la clase True (entidades con el defecto *Data Class*), son iguales al considerar la naturaleza de la entidad.

Sin naturaleza:	Precisión (0,304)	Recuperación (0,5)	F-Measure (0,378)
Con naturaleza:	Precisión (0,304)	Recuperación (0,5)	F-Measure (0,378)

Las medidas de eficiencia cuando utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *Data Class*), mejoran al considerar la matriz de costes para la recuperación y F-Measure, cuyo aumento compensa la disminución de la precisión.

Sin matriz de costes:	Precisión (0,5)	Recuperación (0,286)	F-Measure (0,364)
Con matriz de costes:	Precisión (0,304)	Recuperación (0,5)	F-Measure (0,378)

Las medidas de eficiencia cuando no utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *Data Class*), mejoran al considerar la matriz de costes para la recuperación y F-Measure, cuyo aumento compensa la disminución de la precisión.

Sin matriz de costes:	Precisión (0,467)	Recuperación (0,25)	F-Measure (0,326)
Con matriz de costes:	Precisión (0,304)	Recuperación (0,5)	F-Measure (0,378)

2. En el caso de estudio EclEmma, las medidas de eficiencia cuando utilizamos la matriz de costes, con respecto a la clase True (entidades con el defecto *Data Class*), mejora la precisión y empeoran la recuperación y F-Measure cuando consideramos la naturaleza de la entidad.

Sin naturaleza: Precisión (0,4) Recuperación (0,667) F-Measure (0,5)
 Con naturaleza: Precisión (0,5) Recuperación (0,33) F-Measure (0,4)

Las medidas de eficiencia cuando utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *Data Class*), no mejoran, la F-Measure empeora ligeramente.

Sin matriz de costes: Precisión (1) Recuperación (0,33) F-Measure (0,5)
 Con matriz de costes: Precisión (0,5) Recuperación (0,33) F-Measure (0,4)

Las medidas de eficiencia cuando no utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *Data Class*), empeoran al considerar la matriz de costes.

Sin matriz de costes: Precisión (1) Recuperación (0,667) F-Measure (0,8)
 Con matriz de costes: Precisión (0,4) Recuperación (0,667) F-Measure (0,5)

3. En conclusión, en el caso de estudio EclEmma todos los resultados menos uno son no coincidentes con el resto de los resultados cuando consideramos la naturaleza de la entidad. Sin embargo, teniendo en cuenta que la cantidad de entidades con defectos es muy pequeña y que en JFreeChart, F-Measure nunca empeora, se puede intuir que la matriz de costes es necesaria para detectar el defecto *Data Class*. Los resultados no son claramente concluyentes, por tanto deberían analizarse más aplicaciones con entidades de código que tengan el defecto *Data Class*.

La Tabla 4.28 resume los resultados del análisis: Con o Sin Naturaleza (n, \bar{n}), Con o Sin Costes (c, \bar{c}).

Tabla 4.28. Resumen del análisis del caso de estudio *Data Class* respecto $H_{0,1a}$ y $H_{0,1b}$

	Precisión	Recuperación	F-Measure
$\bar{c}: n$ vs. \bar{n}	No mejora en ningún caso	JFreeChart mejora	JFreeChart mejora
$\bar{n}: c$ vs \bar{c}	No mejora en ningún caso	JFreeChart mejora	JFreeChart mejora
$n: c$ vs \bar{c}	No mejora en ningún caso	JFreeChart mejora	JFreeChart mejora
$c: n$ vs. \bar{n}	EclEmma mejora	No mejora en ningún caso	No mejora en ningún caso

4.9. Evaluación de la eficiencia de métodos de identificación del defecto *Feature Envy*

En esta sección se presenta una réplica del caso de estudio de la Sec. 4.7. La réplica se construye con el objetivo de minimizar la amenaza, referida a cuántos de nuestros resultados se pueden generalizar con otro tipo de defectos. En la réplica las tareas de definición, planificación, operación son las mismas que el caso de estudio original, variando la variable binaria dependiente *God Class* por *Feature Envy*. Por eso en este apartado sólo se presenta el análisis de los resultados.

El objetivo principal de este estudio enunciado en formato GQM [BCR94] es el siguiente:

- *Analizar* entidades de código

- *Con el propósito de estudiar el impacto de su clasificación, según su naturaleza de diseño basada en estereotipos UML*
- *Con respecto a la eficiencia en la identificación de defectos de diseño, en particular el defecto *Feature Envy**
- *Desde el punto de vista de los investigadores*
- *en el contexto académico de la Universidad de Burgos (UBU) y la Universidad de Valladolid (UVa) y de dos aplicaciones de código abierto JFreeChart 1.0.14 y EclEmma 2.1.0.*

Desde el punto de vista de la operación del caso de estudio, la herramienta JDeodorant se bloqueaba al ejecutar la detección de *Feature Envy* sobre la aplicación JFreeChart. La instrumentación de la detección sólo se ha realizado con InCode, por tanto se ha eliminado el objetivo secundario del estudio original. Además se añade en el propósito el impacto de tener en cuenta el coste, tal cómo se analizó en Sec. 4.7.1.

4.9.1. Análisis

Las preguntas que responderemos se derivan del objetivo de este estudio, propuesto en la Sec. 4.9. En función de ellas se plantean las correspondientes hipótesis, como se muestra en la Tabla 4.29.

Tabla 4.29. Preguntas e hipótesis

<p>$H_{0,1}$: ¿Influye la naturaleza de la entidad, modelada como estereotipo UML, en la predicción de defectos de diseño tipo <i>Feature Envy</i>?</p>
<p>$H_{0,1a}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de <i>Feature Envy</i> en el caso de estudio r. Donde, $r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$, $i \in \{ \text{sin naturaleza, con naturaleza} \}$</p>
<p>$H_{0,1b}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de <i>Feature Envy</i> en el caso de estudio r cuando el coste de falsos negativos es k. Donde, $r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$, $i \in \{ \text{sin naturaleza, con naturaleza} \}$, $k \in \{ \text{sin coste, con coste FN según Tabla 4.16} \}$</p>

4.9.1.1. Estudio de la eficiencia de los clasificadores para identificar Feature Envy

En este apartado se estudia la hipótesis $H_{0,1}$ utilizando como medidas de eficiencia de los clasificadores generados en el proceso de aprendizaje, la recuperación, la precisión y F-Measure.

En las Tablas 4.30 y 4.31 se presentan los resultados obtenidos al medir la eficiencia de los clasificadores generados en el proceso de aprendizaje. La interpretación de las tablas ha sido descrita en la Sec. 4.7.2.1. Se ha resaltado con sombreado gris la fila que contiene las medidas derivadas utilizadas para evaluar la eficiencia de los clasificadores en el proceso de aprendizaje. Además hemos considerado necesario mostrar el resto de información para describir el proceso de aprendizaje completo en los casos de estudio.

Tabla 4.30. Evaluación de los clasificadores para el defecto FeatureEnvy sobre el conjunto de datos generado a partir de JFreeChart 1.0.14

		JFreeChart 1.0.14					
i=sin naturaleza j=sin costes	Correctly Classified Instances	12099				99.2779 %	
	Incorrectly Classified Instances	88				0.7221 %	
	Total Number of Instances	12187					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.999	0.807	0.994	0.999	0.996	false
		0.193	0.001	0.5	0.193	0.279	true
	Weighted Avg.	0.993	0.801	0.991	0.993	0.991	
	=== Confusion Matrix ===						
		a	b	<-- classified as			
	12082	17	a = false				
	71	17	b = true				
i=sin naturaleza j=con costes	Correctly Classified Instances	12027				98.6871 %	
	Incorrectly Classified Instances	160				1.3129 %	
	Total Number of Instances	12187					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.992	0.659	0.995	0.992	0.993	false
		0.341	0.008	0.227	0.341	0.273	true
	Weighted Avg.	0.987	0.654	0.99	0.987	0.988	
	=== Confusion Matrix ===						
		a	b	<-- classified as			
	11997	102	a = false				
	58	30	b = true				
i=con naturaleza j=sin costes	Correctly Classified Instances	12098				99.2697 %	
	Incorrectly Classified Instances	89				0.7303 %	
	Total Number of Instances	12187					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.999	0.818	0.994	0.999	0.996	false
		0.182	0.001	0.485	0.182	0.264	true
	Weighted Avg.	0.993	0.812	0.99	0.993	0.991	
	=== Confusion Matrix ===						
		a	b	<-- classified as			
	12082	17	a = false				
	72	16	b = true				
i=con naturaleza j=con costes	Correctly Classified Instances	12036				98.761 %	
	Incorrectly Classified Instances	151				1.239 %	
	Total Number of Instances	12187					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.992	0.625	0.995	0.992	0.994	false
		0.375	0.008	0.256	0.375	0.304	true
	Weighted Avg.	0.988	0.621	0.99	0.988	0.989	
	=== Confusion Matrix ===						
		a	b	<-- classified as			
	12003	96	a = false				
	55	33	b = true				

Tabla 4.31. Evaluación de los clasificadores para el defecto *Feature Envy* sobre el conjunto de datos generado a partir de EclEmma 2.1.0

		EclEmma 2.1.0					
i=sin naturaleza j=sin costes	Correctly Classified Instances	807				99.0184 %	
	Incorrectly Classified Instances	8				0.9816 %	
	Total Number of Instances	815					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.995	1	0.995	0.995	0.995	false
	0	0.005	0	0	0	true	
Weighted Avg.	0.99	0.995	0.99	0.99	0.99		
=== Confusion Matrix ===							
	a	b	←-- classified as				
	807	4	a = false				
	4	0	b = true				
i=sin naturaleza j=con costes	Correctly Classified Instances	806				98.8957 %	
	Incorrectly Classified Instances	9				1.1043 %	
	Total Number of Instances	815					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.994	1	0.995	0.994	0.994	false
	0	0.006	0	0	0	true	
Weighted Avg.	0.989	0.995	0.99	0.989	0.99		
=== Confusion Matrix ===							
	a	b	←-- classified as				
	806	5	a = false				
	4	0	b = true				
i=con naturaleza j=sin costes	Correctly Classified Instances	810				99.3865 %	
	Incorrectly Classified Instances	5				0.6135 %	
	Total Number of Instances	815					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.999	1	0.995	0.999	0.997	false
	0	0.001	0	0	0	true	
Weighted Avg.	0.994	0.995	0.99	0.994	0.992		
=== Confusion Matrix ===							
	a	b	←-- classified as				
	810	1	a = false				
	4	0	b = true				
i=con naturaleza j=con costes	Correctly Classified Instances	806				98.8957 %	
	Incorrectly Classified Instances	9				1.1043 %	
	Total Number of Instances	815					
	=== Detailed Accuracy By Class ===						
		TP Rate	FP Rate	Precision	Recall	F-Measure	Class
		0.994	1	0.995	0.994	0.994	false
	0	0.006	0	0	0	true	
Weighted Avg.	0.989	0.995	0.99	0.989	0.99		
=== Confusion Matrix ===							
	a	b	←-- classified as				
	806	5	a = false				
	4	0	b = true				

$H_{0,1a}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de *Feature Envy* en el caso de estudio r.

Donde,

$r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$,

$i \in \{ \text{sin naturaleza, con naturaleza} \}$

La estructura del análisis para la hipótesis $H_{0,1a}$ es la siguiente:

1. Analizar cómo influye en la eficiencia de la identificación del defecto *Feature Envy* utilizar o no la naturaleza de la entidad, en el caso de estudio JFreeChart 1.0.14.
2. Analizar cómo influye en la eficiencia de la identificación del defecto *Feature Envy* al utilizar o no la naturaleza de la entidad, en el caso de estudio EclEmma 2.1.0.
3. Comparar los resultados anteriores.

1. En el caso de estudio JFreeChart, todas las medidas de eficiencia, con respecto a la clase True (entidades con el defecto *Feature Envy*), empeoran levemente cuando consideramos la naturaleza de la entidad. Los datos que justifican la afirmación son los siguientes:

Sin naturaleza:	Precisión (0,500)	Recuperación (0,193)	F-Measure (0,279)
Con naturaleza:	Precisión (0,485)	Recuperación (0,182)	F-Measure (0,264)

2. En el caso de estudio EclEmma, las medidas de eficiencia, con respecto a la clase True (entidades con el defecto *Feature Envy*), son todas cero con independencia de considerar la naturaleza de la entidad. Una posible razón de este efecto es el escaso número de entidades con el defecto *Feature Envy* (4 de 815).
3. En conclusión, a partir de las observaciones no se puede afirmar que la naturaleza de la entidad ha mejorado las medidas de eficiencia de detección de defectos del tipo *Feature Envy*.

$H_{0,1b}$ Utilizar la naturaleza de la entidad (i) no mejora la eficiencia del método de identificación (clasificación) de *Feature Envy* en el caso de estudio r cuando el coste de falsos negativos es k.

Donde,

$r \in \{ \text{JFreeChart 1.0.14, EclEmma 2.1.0.} \}$,

$i \in \{ \text{sin naturaleza, con naturaleza} \}$,

$k \in \{ \text{sin coste, con coste FN según Tabla 4.16} \}$

La estructura del análisis para la hipótesis $H_{0,1b}$ es la siguiente:

1. Analizar cómo influye en la eficiencia de la identificación de defecto *Feature Envy* utilizar o no la matriz de costes, al utilizar o no la naturaleza de la entidad, en el caso de estudio JFreeChart 1.0.14.
2. Analizar cómo influye en la eficiencia de la identificación de defecto *Feature Envy* utilizar o no la matriz de costes, al utilizar o no la naturaleza de la entidad, en el caso de estudio EclEmma 2.1.0.
3. Comparar los resultados anteriores.

1. En el caso de estudio JFreeChart, las medidas de eficiencia cuando utilizamos la matriz de costes, con respecto a la clase True (entidades con el defecto *Feature Envy*), mejoran al considerar la naturaleza de la entidad.

Sin naturaleza:	Precisión (0,227)	Recuperación (0,341)	F-Measure (0,273)
Con naturaleza:	Precisión (0,256)	Recuperación (0,375)	F-Measure (0,304)

Las medidas de eficiencia cuando utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *Feature Envy*), mejoran al considerar la matriz de costes para la recuperación y F-Measure, cuyo aumento compensa la disminución de la precisión.

Sin matriz de costes: Precisión (0,485) Recuperación (0,182) F-Measure (0,264)
 Con matriz de costes: Precisión (0,256) Recuperación (0,375) F-Measure (0,304)

Las medidas de eficiencia cuando no utilizamos la naturaleza de la entidad, con respecto a la clase True (entidades con el defecto *Feature Envy*), empeoran al considerar la matriz de costes para la precisión y F-Measure, a pesar de la mejora de la recuperación.

Sin matriz de costes: Precisión (0,5) Recuperación (0,193) F-Measure (0,279)
 Con matriz de costes: Precisión (0,227) Recuperación (0,341) F-Measure (0,273)

2. En el caso de estudio EclEmma, en todos los valores de las medidas de eficiencia son cero con independencia del valor de las variables naturaleza y costes. Una posible razón de este efecto es el escaso número de entidades con el defecto *Feature Envy* (4 de 815).
3. En conclusión, el caso de estudio EclEmma no aporta resultados relevantes. En el caso de JFreeChart se observa que la matriz de costes es necesaria para detectar el defecto *Feature Envy*. Los resultados mejoran cuando se combinan el coste y la naturaleza de la entidad.

La Tabla 4.32 resume los resultados del análisis: Con o Sin Naturaleza (n , \bar{n}), Con o Sin Costes (c , \bar{c}).

Tabla 4.32. Resumen del análisis del caso de estudio *Feature Envy* respecto $H_{0,1a}$ y $H_{0,1b}$

	Precisión	Recuperación	F-Measure
\bar{c} : n vs. \bar{n}	No mejora en ningún caso	No mejora en ningún caso	No mejora en ningún caso
\bar{n} : c vs \bar{c}	No mejora en ningún caso	JFreeChart mejora	No mejora en ningún caso
n : c vs \bar{c}	No mejora en ningún caso	JFreeChart mejora	JFreeChart mejora
c : n vs. \bar{n}	JFreeChart mejora	JFreeChart mejora	JFreeChart mejora

4.10. Conclusiones de la experimentación

El objetivo principal del estudio empírico, presentado con los casos de estudio de las Sec. 4.7, 4.8 y 4.9, es comprobar si la naturaleza de la entidad de código, entendida como estereotipos estandar de UML, mejora la eficiencia de los métodos de identificación de defectos de diseño basados en métricas de código. Este objetivo es de interés porque las técnicas de identificación de defectos en entidades de código son útiles para mejorar el mantenimiento del mismo.

Durante el estudio se ha observado que en el problema de identificación de defectos sobre entidades de código, se generan conjuntos de datos no balanceados. Para resolver este problema, hemos considerado el coste del error de los falsos negativos en los métodos de detección.

En la Tabla 4.33 se muestra un resumen global de los resultados obtenidos en el estudio. La interpretación de las celdas que contienen el texto “Significativo” indica que la variable o combinación de variables (primera fila de la tabla) mejoran las heurísticas de detección del defecto de diseño asociado (la primera columna).

Tabla 4.33. Resumen de los resultados del estudio empírico

	Naturaleza	Costes	Naturaleza y Costes
<i>God Class</i>	Significativo	Significativo	Significativo
<i>Data Class</i>	No Significativo	No Significativo	No Significativo
<i>Feature Envy</i>	No Significativo	Significativo	Significativo

Podemos concluir lo siguiente:

1. En el estudio de la eficiencia de los clasificadores para identificar el defecto *God Class*, los resultados observados indican que la naturaleza de diseño de la entidad mejora su eficiencia. Además, la matriz de costes, cuando se penaliza el coste de los falsos negativos, también mejora dichas medidas. Por ello, la matriz de costes y la naturaleza de diseño de la entidad son dos factores relevantes para mejorar dicha eficiencia.
2. En el estudio de comparación de los dos métodos de identificación de *God Class*, los resultados indican que no existe concordancia, por tanto la clasificación tiene un grado de subjetividad. Es por ello por lo que se recomienda que los métodos de identificación de *God Class* sean adaptados al contexto de la aplicación con técnicas de aprendizaje supervisado.
3. En el estudio de la eficiencia de los clasificadores para identificar el defecto *Data Class*, no se observa claramente que la matriz de costes y la naturaleza de diseño de la entidad sean dos factores relevantes para mejorar dicha eficiencia, aunque en este caso el número de clases con este defecto es muy pequeño, lo que ha podido influir en el resultado no significativo.
4. En el estudio de la eficiencia de los clasificadores para identificar el defecto *Feature Envy*, se observa claramente que la matriz de costes mejora la eficiencia. Por otro lado, aunque la naturaleza de la entidad de manera aislada no mejora las medidas de eficiencia, en combinación con la matriz de costes se obtienen las mejores medidas de eficiencia.

FRAMEWORK PARA LA MEDICIÓN Y LA GESTIÓN DE DEFECTOS

En capítulos anteriores se ha empleado la medición como técnica fundamental. En la experimentación realizada en el Cap. 3 se han utilizado distintos conjuntos de métricas proporcionados por las herramientas RefactorIt y SourceMonitor. Hemos observado que las métricas de código que forman parte de un proceso de medición, deben cumplir dos condiciones:

- Deben formar un conjunto abierto, permitiendo incorporar nuevas métricas
- Deben ser independientes del lenguaje de programación

En la caracterización de las herramientas que automatizan el proceso de medición, se ha observado que cada una proporciona su solución particular, sin afrontar estos dos problemas de diseño: facilitar la inclusión de nuevas métricas y reutilizar la definición de la métrica con independencia del lenguaje. Con el propósito de resolver estos problemas, en este capítulo se propone el diseño de un *framework* [FSJ99] que calcula métricas.

La propuesta de detección de defectos que hemos realizado en esta tesis se basa en técnicas de clasificación de minería de datos. Nuestras definiciones de defectos de diseño incluyen clasificadores binarios, que se generan a partir de históricos de medidas de métricas de código y de la información acerca de la naturaleza de diseño de la entidad. A partir de las definiciones se automatiza la tarea de identificación de defectos. En consonancia con esta propuesta, se define un *framework* de gestión de defectos de diseño, cuyo objetivo es permitir al usuario añadir nuevos defectos, modificarlos o borrarlos.

Los procesos deben estar asistidos por herramientas que automaticen sus tareas. Como prueba de concepto, en este capítulo, se presentan tres prototipos de herramientas, uno relacionado con la modificación del proceso de medición especificado en el Cap. 3. Los otros dos, relacionados con el proceso de gestión de defectos especificado en el Cap. 4. Una versión de los prototipos está disponible en [Lóp12b]. El objetivo del prototipado es poder observar la interacción del usuario con las actividades de los procesos propuestos.

5.1. Framework de soporte para el cálculo de medidas

Las soluciones actuales para el cálculo de métricas son particulares a algún lenguaje de programación concreto. Pese a que es una solución válida, se debe recordar que la mayoría de las métricas, y en mayor medida las orientadas a objetos, son independientes del lenguaje.

Lo que inicialmente parece 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. También el metamodelo recoge información general sobre instrucciones, asignaciones y expresiones que es necesaria para el cálculo de métricas como $V(G)$ [McC76], WMC [CK94], etc.

El metamodelo MOON [Cre00, LC03, LMC03], utilizado en nuestro caso, mantiene dicha información. Tomando el metamodelo MOON, o bien otros metamodelos similares como los de FAMIX [DTS99], se 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 [FSJ99].

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, MOON en este caso, 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.

5.1.1. Recorrido sobre los elementos del metamodelo

Para evitar tener que modificar todas las clases que modelan los elementos “medibles”, incluyendo una operación nueva, se aplica el patrón de diseño *Visitor* [GHJV95]. 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 diferentes propiedades de ciertos elementos del modelo.

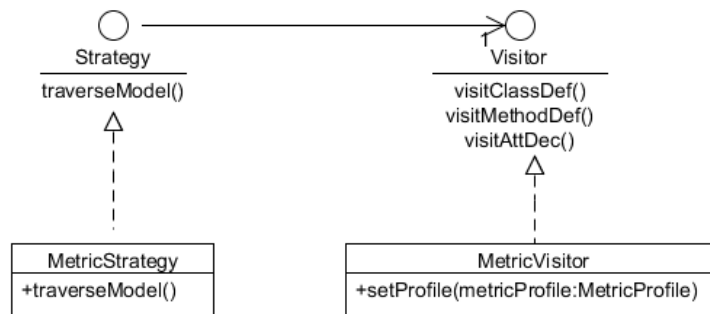
Aunque inicialmente pueda parecer un esfuerzo no justificable, introducir dicho patrón favorece la posibilidad futura de poder realizar otras operaciones sobre dichos elementos.

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, de medición en este caso (ver Fig. 5.1).

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 5.1. Núcleo del motor para la 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* [GHJV95], se pueda elegir de manera dinámica el algoritmo particular que conviene en cada caso para acceder a todas las instancias del metamodelo.

5.1.2. Jerarquía de métricas

Las métricas han sido clasificadas de distintas formas. En [LA97] se recogen distintas clasificaciones más o menos complejas. En este caso optamos por una clasificación simple, centrándonos en los niveles de granularidad (sistema, clase y métodos).

Las métricas relativas a atributos, están ligadas a la clase como contenedora de dichas propiedades. Por ejemplo la métrica NOA (Number Of Attributes) ¹ [LK94] 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 de las clases del metamodelo. En el caso del metamodelo MOON, la pérdida de información se reduce a las sentencias condicionales y bucles, que son almacenadas sin contenido semántico dentro del metamodelo.

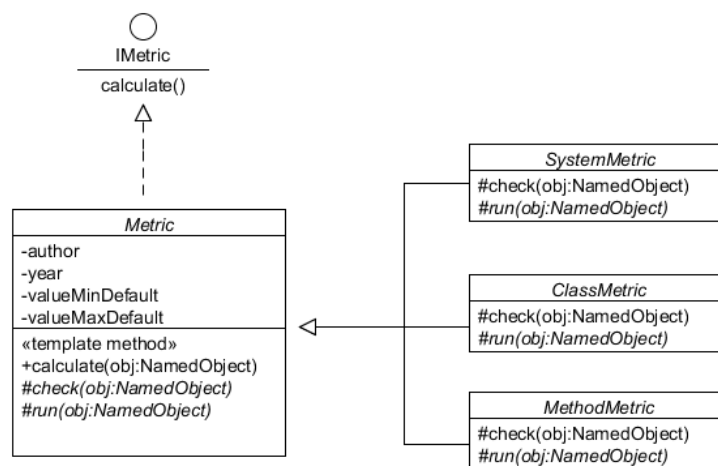
Esto impide, por ejemplo, poder definir las métricas relacionadas con la complejidad ciclomática de McCabe [McC76] a nivel de metamodelo. Este problema se aborda con una solución de framework [FSJ99]. El enfoque seguido respecto a la falta de información, muchas veces ocasionado por dependencia del lenguaje, es utilizar un lenguaje

¹También conocida como NOF (Number of Fields)

modelo intermedio MOON, que recoge las características comunes de distintas familias de lenguajes OO. Cada lenguaje en particular tiene su propia extensión definida a partir de MOON, donde se recogen sus características específicas. De esta forma se pueden calcular la métricas 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 métricas establecida se presenta en la Fig. 5.2. El método `calculate()` de la clase abstracta `Metric` cumple el rol de *Template Method* [GHJV95]. El método plantilla realiza una comprobación (método `check`) que verifica si se puede aplicar la métrica sobre el objeto pasado como parámetro. Si la comprobación es correcta, se ejecuta la medición (método `run`). Las comprobaciones se definen en el núcleo del framework, mientras que las ejecuciones concretas son parte de cada extensión, siguiendo el patrón *Command* [GHJV95]. Ambos métodos, `check` y `run` conforman la plantilla de ejecución al invocar al método `calculate`.

Figura 5.2. Núcleo de clases del framework de métricas



Para la recolección de las medidas obtenidas se ha utilizado el patrón *Collecting Parameters* definido en [Bec97]. Se implementa a través de la clase `MetricResult` (ver Fig. 5.3). 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`. Con esto se obtiene un comportamiento similar a la utilización de una pizarra donde se apuntan los resultados colectivos por parte de todos los objetos que colaboran.

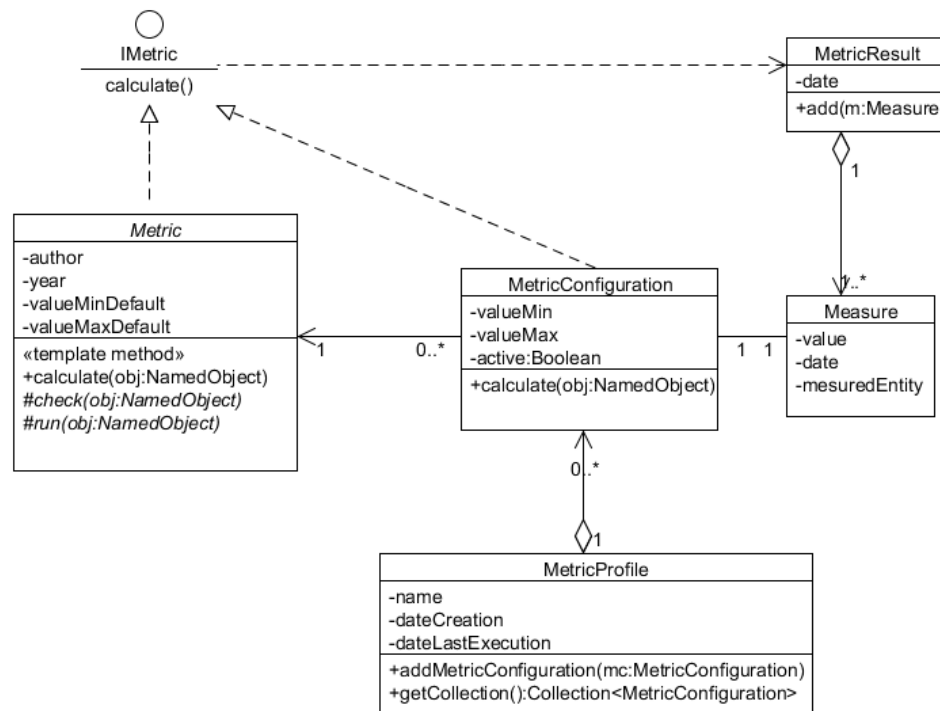
5.1.3. Personalización de las métricas: Perfiles

En el Cap.3 Sec. 3.1 se ha observado empíricamente que las métricas plantean ciertos problemas a la hora de aplicarlas e interpretarlas de manera absoluta. 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 y teniendo en cuenta la naturaleza de diseño de la entidad medida (ver Sec. 3.4.4 y Sec. 3.5.4).

Para ello se crea una clase de envoltorio, **MetricConfiguration** (ver Fig. 5.3), 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.

Figura 5.3. Personalización del cálculo con perfiles

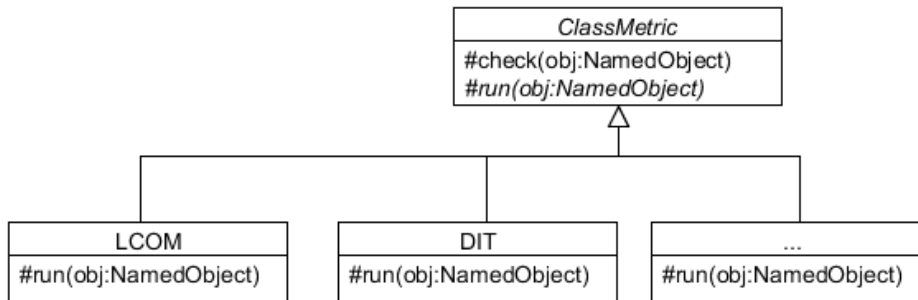


Mediante la agrupación de distintas configuraciones en la clase `MetricProfile` (ver Fig. 5.3), se posibilita la definición de distintos perfiles por parte del programador. Como se ha observado empíricamente en el Cap. 3, dependiendo del dominio en el que se desarrolle los valores cambiarán.

5.1.4. Cálculo de las medidas

Colocando todas las piezas juntas, el proceso concreto de recorrido y cálculo se inicia por medio de una estrategia de recorrido 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 calculan aquellas métricas configuradas en el perfil que se

Figura 5.4. Ejemplo de extensión concreta del framework

está aplicando. Los resultados obtenidos para cada objeto instanciado sobre el metamodelo, se recogen como una instancia de *Measure*, que permite trazabilidad hacia la métrica concreta con la que se ha obtenido a través de *MetricConfiguration*, y hacia la entidad sobre la que ha sido calculada. Las medidas se agrupan en lo que se denomina el “resultado de métricas” (clase *MetricResult*) para posibilitar el posterior análisis y presentación de resultados.

5.1.5. Ejemplo de validación del framework

El framework ha sido implementado sobre el metamodelo MOON ya definido. Dicho metamodelo 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 LCOM(Lack Of Cohesion) [CK94] 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. 5.4). El cuerpo del método *run* se redefine para obtener un valor numérico correspondiente a partir de la información extraída del modelo. Este valor se representa 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 desarrollador 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.

5.1.6. 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 orien-

tados a objetos muy amplio, tantos como parsers hacia el metamodelo se construyan. En nuestro trabajo, tenemos implementado un metamodelo (MOON [LMC03]) y contamos con parsers hacia el metamodelo para Java y Eiffel. Actualmente se encuentra en desarrollo un parser para C#.

Dentro de las posibles mejoras al framework, se apunta:

- La introducción del patrón *Observer* [GHJV95] para actualizar y recalculer sólo las métricas asociadas a las entidades modificadas.
- La introducción de filtros adicionales para indicar que ciertas métricas no se aplican por igual a un cierto tipo de elemento. Por ejemplo, los valores máximos de la métrica NOP (Number Of Parameters) [PJ88] pueden relajarse cuando se trata de métodos especiales de tipo constructores, en los lenguajes que lo permitan.
- Incorporar una capa de presentación gráfica para la asistencia a la interpretación de valores obtenidos.

5.2. Prototipo de herramienta adaptada al proceso de medición propuesto

En esta sección se aborda una adaptación de una herramienta existente de cálculo de métricas de código que permite aplicar el nuevo proceso de medición propuesto en la Sec. 3.2.

El prototipo propuesto no utiliza el framework definido en la Sec. 5.1. En el momento de su definición, no se disponía de analizadores completos de código (*parsers*), para algún lenguaje de programación, que almacenara toda la información del código sobre instancias del metamodelo MOON y sus extensiones. Esta información es necesaria para el cálculo de métricas. Consideramos que el desarrollo completo de este tipo de analizadores de código se sale fuera del ámbito de esta tesis.

La incorporación de las nuevas tareas propuestas en el proceso implican incorporar dos nuevos requisitos en las herramientas: por un lado, la gestión de la clasificación de entidades de código en categorías y, por otro, los valores umbrales de las métricas asociados a cada categoría considerada. El primero lleva asociado la definición de una clasificación y algoritmos de clasificación de entidades de código en categorías (ver Alg. 1). La herramienta permitirá evaluar entidades de código usando los valores recomendados que indique su categoría. El segundo, implica un proceso de recogida de datos que permita obtener unos valores umbrales por categoría.

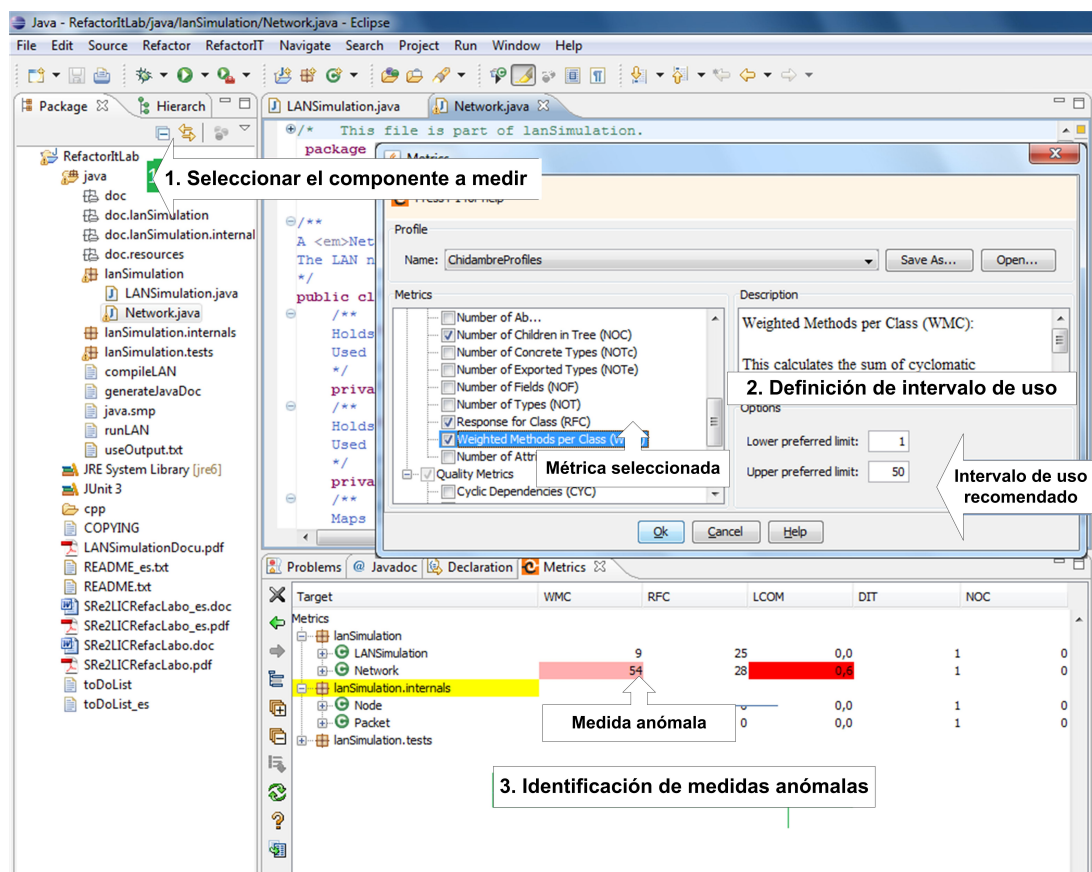
5.2.1. Adaptación de la herramienta RefactorIt

La adaptación del nuevo proceso de medición necesita de herramientas que asistan estas nuevas actividades. En este caso se ha optado por extender la herramienta RefactorIt [Aqr02]. Es una herramienta open source que sirve para inspeccionar código Java mediante métricas y reglas semánticas de código. RefactorIt cuenta con varias formas de

distribución, como herramienta de escritorio o como plugin de Eclipse. Además, proporciona un catálogo de refactorizaciones que facilitan el proceso de mantenimiento.

Actualmente la herramienta tiene automatizado el proceso básico de medición. En la Fig. 5.5 se muestra una captura de pantalla con el resultado de la evaluación de un proyecto de Eclipse llamado RefactorItLab y documentado en [DDN02]. En los rectángulos numerados se indican las partes de la interfaz gráfica que sirven como entrada a las actividades básicas del proceso de medición. Además, se destaca con los rectángulos no numerados, el resultado de una evaluación respecto a la métrica WMC (*Weighted Methods per Class*) y la identificación de una medida anómala, en la clase *Network*, respecto a los valores umbrales recomendados por la herramienta [1-50].

Figura 5.5. Proceso de medición con la herramienta RefactorIt



En los siguientes apartados se analizan los nuevos requisitos que debe incorporar una herramienta para adaptarse al nuevo proceso de medición propuesto en la Sec. 3.2 y la adaptación concreta de la herramienta RefactorIt, que será referenciada como RefactorItUBU.

5.2.1.1. Clasificación de entidades de código

Las entidades de código pueden necesitar diferentes valores umbrales para detectar anomalías dependiendo de ciertas clasificaciones, como la que se ha tenido en cuenta, en los capítulos anteriores, para informar sobre la naturaleza de diseño de la entidad. Las herramientas deben proporcionar algún mecanismo, automático o manual, que permita al inspector clasificar las entidades en las categorías que compongan la clasificación.

Aunque existen clasificaciones que pueden considerarse como estándar es preferible que las herramientas permitan al inspector definir sus propias clasificaciones. En este caso, se ha partido de la clasificación de entidades de código utilizada en la experimentación empírica presentada en el Cap. 3. La clasificación se basa en la naturaleza de diseño de la entidad, que es expresada con el uso de estereotipos estándar sobre clasificadores de UML: *Exception*, *Interface*, *Entity*, *Control*, *Test*, *Utility* y *UnKnown*.

Se ha añadido a la herramienta la nueva funcionalidad, que corresponde a la creación de una clasificación, definiendo un fichero de configuración de donde se extraen las diferentes categorías consideradas (`refactoritubu.estereotipos.csv`). Esta clasificación será utilizada en dos actividades posteriores: una cuando se definen los valores umbrales de cada métrica y otra cuando se realiza la medición del componente. También se utiliza con la interfaz gráfica.

Las siguientes figuras muestran de manera concreta la funcionalidad añadida en RefactorItUBU. Suponemos el siguiente contenido del fichero con la siguiente clasificación de categorías: *Unknown*, *Exception*, *Interface*, *Control*, *Entity*, *Test*, *Utility*. La Fig. 5.6 muestra la nueva definición de valores umbrales de cada métrica, en la parte inferior derecha se añade un panel etiquetado con cada uno de los estereotipos y los valores umbrales que se consideran recomendables para cada uno. Al igual que ocurría en la funcionalidad original de la herramienta, estos valores umbrales deben de ser introducidos por el usuario y pueden ser almacenados en ficheros de profile (ver Sec. 5.1.3).

Por último, en la Fig. 5.7 se muestra la evaluación de entidades donde se permite al usuario definir el estereotipo de cada entidad. Desde el punto de vista del inspector, es interesante destacar que si la clasificación no es cerrada se debe considerar una categoría para las entidades de clasificación desconocida.

5.2.1.2. Clasificación de entidades en las categorías consideradas

Cuando se trabaja con sistemas reales el número de entidades de código a clasificar es muy grande. Es deseable que el inspector disponga de facilidades que le ayuden a realizar esta nueva actividad de una manera eficiente. Además se parte de la hipótesis de que la clasificación puede ser subjetiva y es el inspector el encargado de tomar la decisión final sobre cómo clasificar las entidades de código. En este sentido, la herramienta podría proporcionar dos nuevas funcionalidades:

- clasificación por agrupamiento de entidades
- clasificación automática

Figura 5.6. Definición de valores umbrales para cada métrica y categoría con RefactorItUBU

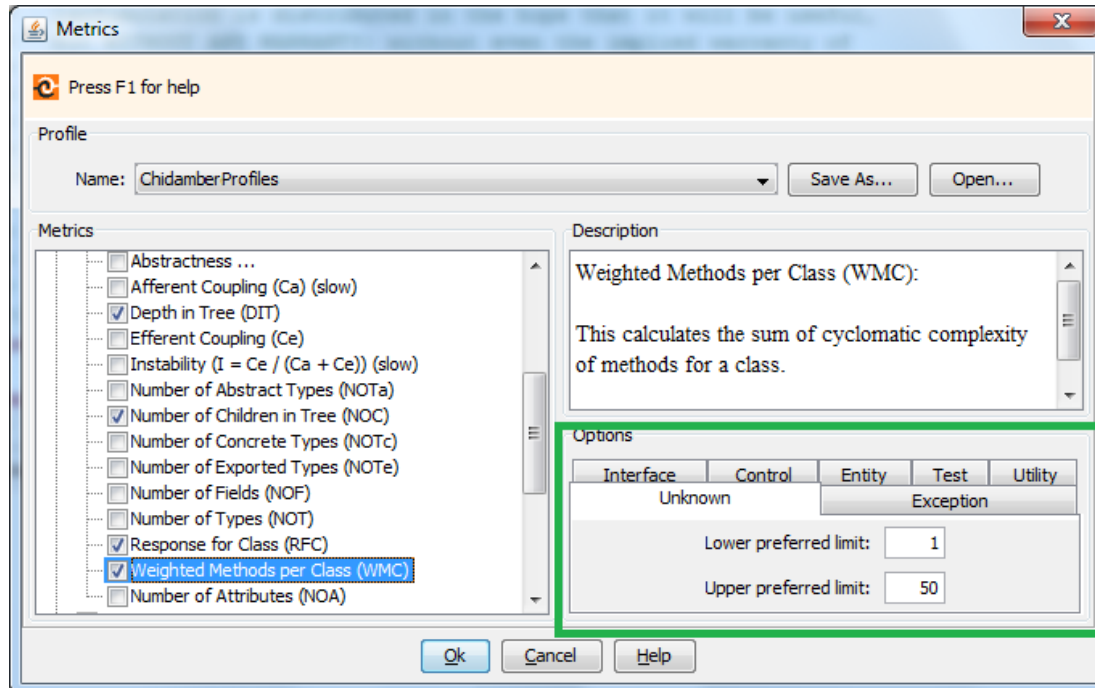


Figura 5.7. Inspección de entidades de código con RefactorItUBU

Target	STR	WMC	RFC	DIT	NOC	LCOM
Metrics	Unknown					
lanSimulation	Unknown					
LANSimulation	Unknown	9	25	1	0	0,0
Network	Unknown	54	28	1	0	0,6
lanSimulation.internals	Unknown					
Node	Unknown	2	0	1	0	0,0
Packet	Unknown	2	0	1	0	0,0
lanSimulation.tests	Test					
LANTests	Test	38	39	3	0	0,0
LANTests\$PreconditionViolationTestCase	Test	2	4	4	0	0,0

La arquitectura de la aplicación (capas, componentes) hace que las entidades de código tengan algún tipo de agrupación lógica que debe ser identificada por el inspector. Además, esa organización lógica tiene una correspondencia con la estructura física a través de las propias entidades de código. Las estructuras de agrupación física son: por un lado los paquetes, contienen paquetes y clases, por otro lado las clases, contienen métodos. La aplicación de una categoría sobre una estructura de agrupamiento es propagada sobre el resto de sus componentes. Así, una aplicación con un agrupamiento lógico marcado por una arquitectura de tres capas podría clasificarse indicando la ca-

tegoría a los tres paquetes que contienen los niveles superiores de la arquitectura. Cada cambio de categoría obliga una nueva evaluación de la entidad con los valores umbrales del nuevo estereotipo elegido.

Un inspector de código querría, además, funcionalidad para poder disponer de métodos de clasificación automática que puedan ser posteriormente ajustados por él. En este sentido, cabe destacar como técnica de identificación de entidades, las distintas convenciones de nombres que utilizan los programadores y arquitectos software. Por ejemplo, las entidades de código cuyos nombre contienen las cadenas “interface”, “gui”, “form”, etc suelen pertenecer a la categoría *Interface*. El conocimiento necesario basado en convención de nombres para identificar entidades puede ser genérico respecto a convenciones de diseño o del lenguaje de programación, o bien por conocimiento específico de los requisitos del proyecto. Por ejemplo, si en un proyecto de cálculo de métricas aparece una capa llamada *metric*, en una primera inspección podría pertenecer a las categorías *Entity* o *Control*. Las convenciones de las bibliotecas y del propio lenguaje de programación pueden ayudar a identificar entidades de manera inequívoca, este es el caso de los test de JUnit y las excepciones de Java.

La extensión que se ha hecho de RefactorIt ha incorporado estas funcionalidades. El cambio de categoría sobre una entidad de código de agrupamiento se propaga sobre el resto de entidades que contiene. Para realizar ese cambio desde la interfaz de usuario (ver Fig. 5.7) es necesario elegir la nueva categoría de la lista desplegable.

El algoritmo de clasificación automática lee de un fichero de configuración las convenciones de nombres asociadas a cada categoría considerada. De esta manera, el inspector puede incidir en el algoritmo e introducir convenciones específicas del contexto de la aplicación que está inspeccionando. La especificación estará compuesta por una cuádrupla (convención paquetes, estereotipo paquete, convención clases, estereotipo clase). Por ejemplo, para aplicar la convención de nombres propuesta por JUnit se indica con la siguiente cuádrupla (“test”, “Test”, “Test”, “Test”), significa que los paquetes que contengan en su nombre la cadena test serán clasificados con la categoría de Test y que las clases que contengan en su nombre la cadena Test serán clasificadas con la categoría de Test. Las excepciones pueden no estar agrupadas en paquetes, por tanto para identificar excepciones dentro del paquete test se necesita añadir la cuádrupla (“test”, “Test”, “Exception”, “Exception”). Otro ejemplo es la cuádrupla (“ui”, “Interface”, “listener”, “Control”), donde por defecto se clasifican todas las entidades que componen un paquete con la cadena “ui” con el estereotipo “Interface”, excepto las clases que contienen la cadena “listener” que se clasifican como “Control”. Este algoritmo es ejecutado cada vez que desde la interfaz de usuario se manda realizar la medición de un componente. Una vez aplicada la clasificación sobre las entidades de código, se utilizan múltiples intervalos de valores recomendados, condicionados a cada categoría.

5.3. Framework de soporte para la gestión de defectos

En la sección de experimentación del Cap. 4 se ha observado que las reglas de detección de defectos son diferentes dependiendo de la aplicación. Las soluciones actuales para la detección de defectos de diseño no permiten su adaptación a diferentes contextos de desarrollo. La funcionalidad básica que proporcionan al inspector es la selección de un conjunto de defectos, de entre los posibles, y, como resultado, se retorna una tabla de entidades de código que presentan los defectos elegidos.

Algunas soluciones permiten modificar los valores umbrales (absolutos vs. relativos) de las reglas que especifican la definición del defecto. Pero en ningún caso se permite al inspector validar los resultados de la detección con indicadores de eficiencia de la detección (recuperación vs. precisión).

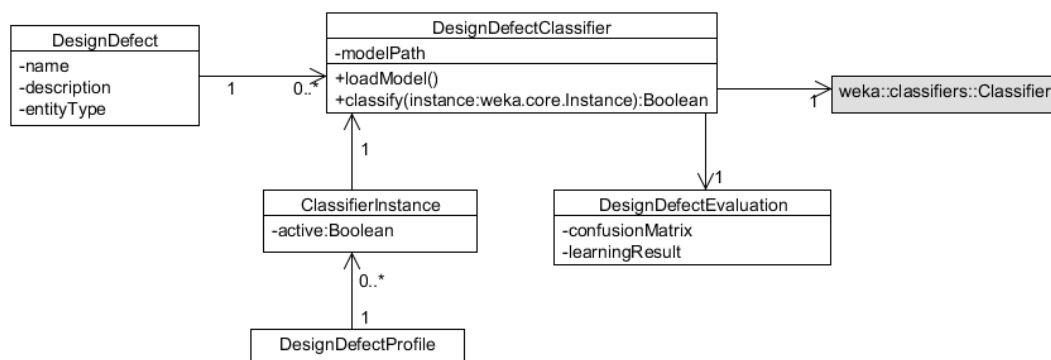
Nuestra solución a este problema se basa en la utilización de aprendizaje supervisado mediante algoritmos de clasificación implementados en la biblioteca Weka de minería de datos. Las instancias de la clasificación son valores de medidas de código de la entidad y la clase binaria (true o false) es la predicción de la existencia del defecto.

En esta sección se propone un framework que pueda utilizarse y adaptarse tanto con defectos de diseño ya definidos como con nuevos defectos. La propuesta se realiza desde el punto de vista del diseño, pero con la dependencia sobre el sistema Weka de minería de datos.

5.3.1. Definición de defectos de diseño

En esta sección se identifican, relacionan y se describen las abstracciones necesarias para definir defectos de diseño. En la Fig. 5.8 se muestra un diagrama de clases donde se identifican y se relacionan estas abstracciones.

Figura 5.8. Núcleo de la gestión de defecto de diseño



Los defectos de diseño (`DesignDefect`) se describen mediante un nombre, una definición

textual y el tipo de entidad de código (subsistema, paquete, clase, método) donde pueden aparecer.

Aunque se puede crear un defecto de diseño sin tener asociado un clasificador, para la tarea de identificación es necesario que tenga asociado un clasificador o varios. Para cargar el clasificador se utiliza el método `loadModel()` de la clase `DesignDefectClassifier`. Una vez cargado el clasificador se puede comprobar si una entidad de código tiene el defecto invocando al método `classify()`. La instancia necesaria para clasificar (`DesignDefectClassifier.classify(weka.core.Instance)`) se crea con el nombre de la entidad (`Entity.name`), sus medidas (`metricframework::Measure`) y su naturaleza de diseño (`Entity.stereotype`) (ver clase `Entity` en Fig. 5.9).

`DesignDefect` y `DesignDefectClassifier` son las abstracciones básicas del framework, a partir de las cuales se deben generar nuevas instancias con los defectos que se quieran gestionar.

Cada clasificador tiene asociada una evaluación (`DesignDefectEvaluation`), representada por la matriz de confusión y las medidas de precisión y recuperación calculadas en el proceso de aprendizaje. Estos indicadores sirven para poder valorar la eficiencia del clasificador. A partir de un clasificador inicial, el inspector puede crear nuevos clasificadores incluyendo nuevas instancias en el conjunto de datos históricos. En este caso es importante que el inspector haga uso de los indicadores de eficiencia para disponer de un criterio de validación de cada nuevo clasificador.

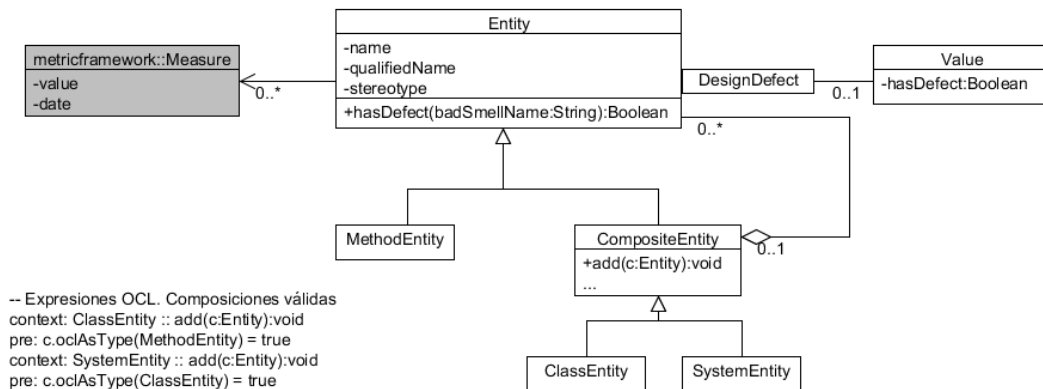
Se puede definir un perfil `DesignDefectProfile` para agrupar varios defectos y poder realizar en un sólo paso la tarea de identificación. Un perfil de defectos asocia de manera única un clasificador activo `ClassifierInstance` por cada defecto.

5.3.2. Integración con el framework de métricas

Las entidades de código, representadas por instancias de la clase `Entity`, son los objetos sobre los cuales se quiere conocer si tienen o no el defecto. Las entidades pueden ser sistemas, clases o métodos. Los sistemas contienen clases y las clases contienen métodos. Esta clasificación de entidades se ha modelado con el patrón de diseño *Composite* [GHJV95] (ver el diagrama de clases de la Fig. 5.9).

Las entidades se describen por su nombre simple y cualificado. Además se incluyen información sobre su naturaleza de diseño mediante el atributo `stereotype`. Aunque en el diagrama de clases el nombre cualificado `qualifiedName` y el estereotipo se han considerado atributos, ambos podrían ser calculados. El nombre cualificado se puede calcular concatenado los nombres simples de los compuestos donde está incluido (`CompositeEntity`). Desde un punto de vista del patrón de diseño *Composite*, se debe definir la operación abstracta en el componente (`Entity`), y redefinirla tanto en los componentes simples concretos (`MethodEntity`) como en los componentes compuestos concretos (`CompositeEntity`).

El estereotipo se puede calcular aplicando el Algoritmo 1. Dicho Algoritmo, toma como

Figura 5.9. Relación defectos de diseño con valores de medidas

entrada el nombre cualificado de la entidad, para inducir el estereotipo. El nombre debería estar definido en la clase `Entity`,

En este framework de gestión de defectos, las instancias de clasificadores son las encargadas de detectar los defectos (`DesignDefectClassifier`). El clasificador se obtiene a partir de instancias, que son un conjunto de valores numéricos asociados a cada entidad. Estos valores numéricos se corresponden con los valores de las medidas de métricas de código de la entidad. En el framework de métricas expuesto en la Sec. 5.1 se corresponde con instancias de la clase `Measure`. Este es el punto de conexión de ambos frameworks.

5.3.3. Puntos fuertes y débiles de la propuesta

El principal beneficio de este planteamiento es que una vez implementado el framework, se dispone de una herramienta que puede ser extendida por un usuario para crear o modificar nuevos defectos de diseño.

Las principales fortalezas de esta propuesta se resumen en los siguientes puntos:

- Se permite una gestión de defectos (creación, modificación y borrado) a partir de instancias de entidades especificadas por el inspector.
- Un defecto de diseño puede estar definido con varios clasificadores. De esta forma se pueden ajustar los árboles de decisión (clasificadores) en función de la aplicación concreta. Esta funcionalidad se ha incorporado como consecuencia de lo observado en la experimentación del Cap. 4.
- El inspector dispone de indicadores, como la precisión y recuperación, que le ayudan a validar los nuevos clasificadores.
- Los clasificadores tienen en cuenta la información relacionada con la naturaleza

de diseño de la entidad recogida como estereotipos UML. En el Cap. 4 se ha observado que esta información es significativa.

El framework depende en gran medida de un buen conjunto de datos de entidades con defecto, para generar clasificadores precisos y con alta recuperación. Esta tarea no es trivial, necesita de una fase experimental que obliga a los inspectores a disponer de habilidades relacionadas con la experimentación y minería de datos. El framework se podría mejorar si proporcionara una funcionalidad para generar un conjunto inicial de datos para cada defecto de manera deductiva. Es decir, que mediante reglas lógicas basadas en métricas, como las propuestas en [LM06], se generaran instancias ficticias. A partir de ese conjunto inicial el inspector podría añadir más instancias para generar nuevos clasificadores.

5.4. Prototipos de herramientas adaptados al proceso de gestión de defectos

En esta sección se presentan dos prototipos de herramientas que automatizan parcialmente las tareas del proceso de gestión de defectos de diseño descrito en la Sec. 4.1: definición, experimentación, identificación y validación.

En las herramientas estudiadas en el estado en el estado de arte (ver Cap. 2) se ha observado que todas abordan, con algún tipo de técnica, la tarea de identificación. Sin embargo, ninguna de las estudiadas asisten a la validación de las reglas de detección que utilizan. En nuestro caso, ambos prototipos abordan la tarea de identificación y de validación. En el segundo se abordan todas las tareas del proceso.

Ambos prototipos tienen en cuenta la información relacionada con la naturaleza de diseño de la entidad, recogida como estereotipos UML. En el Cap. 4 se ha observado que esta información es significativa.

5.4.1. Defectos arquitectónicos

Architecture Defects Finder (ADF) es un plugin para Eclipse, que automatiza las tareas de identificación y validación del proceso descrito en la Sec.4.1. Este prototipo fue presentado en un taller de herramientas en las XVI Jornadas de Ingeniería del Software y Bases de Datos [LCMM11].

ADF permite al usuario realizar las siguientes operaciones:

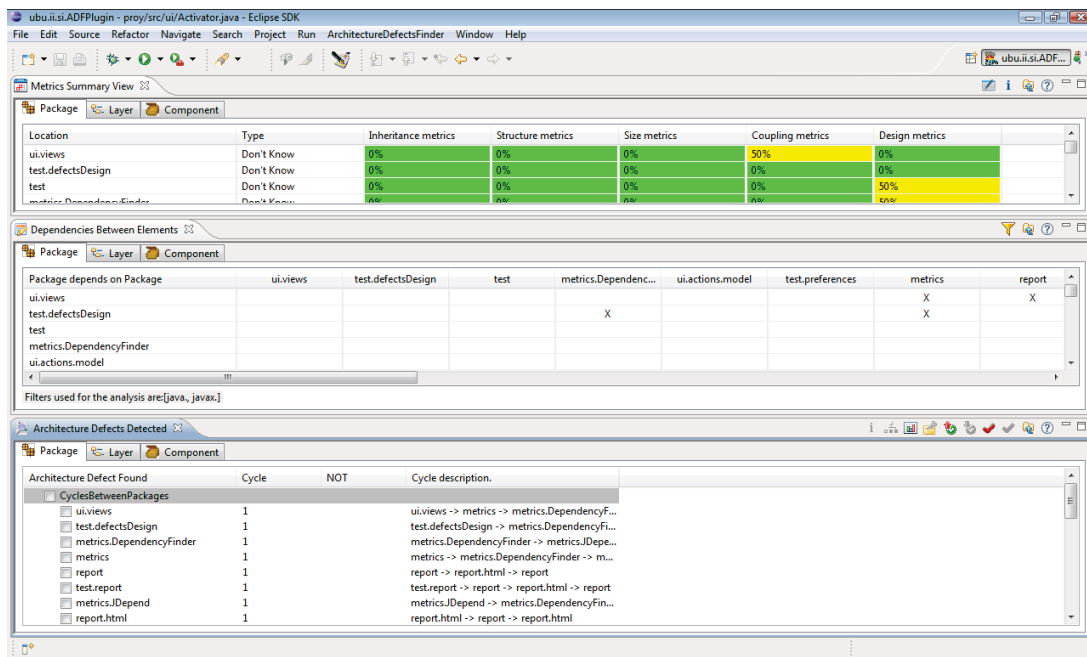
- Calcular y analizar métricas de código sobre las entidades: paquetes, capas y componentes, que componen la aplicación a inspeccionar.
- Clasificar las entidades en estereotipos UML [JBR00]: *entity*, *user interface*, *device interface*, *system interface*, *utility*, *test*, *control*.

- Detectar posibles defectos de arquitectura, a nivel de paquete, capa o componente [SR06], aplicando técnicas de minería de datos sobre métricas de código y teniendo en cuenta la información del estereotipo UML de la entidad.

Los defectos arquitectónicos detectados son: *entidad no usada*, *ciclos entre entidades*, *entidad demasiado pequeña*, *entidad demasiado grande*. Estos defectos relacionados con las entidades del sistema, fueron presentados en [SR06].

La integración con Eclipse se ha basado en: obtener la información del sistema a evaluar desde la definición del proyecto, la creación de nuevos menús de acceso a la nueva funcionalidad, definición de una nueva perspectiva con tres vistas y la integración de la ayuda en el entorno. En la Figura 5.10 se muestra una captura de pantalla de la perspectiva creada con las tres vistas: resumen de métricas, dependencia entre entidades y detección de defectos. Cada vista contiene un panel etiquetado con cada una de las entidades paquete, componente, capa.

Figura 5.10. Perspectivas del plugin ADF



El plugin utiliza los módulos de cálculo de métricas de las herramientas software *DependencyFinder* [Tes] y *JDepend* [Con99]. Estos módulos se han adaptado incorporando los conceptos de capa y componente, y proporcionando filtros de las entidades a analizar. Además, ADF integra cierta funcionalidad de la herramienta de minería de datos Weka [Wai07], en concreto, la implementación del algoritmo de clasificación J48, la representación gráfica del árbol de clasificación que se genera y el cálculo de la medida *F-Measure* para validar el proceso de detección.

5.4.2. Gestión integral de defectos de diseño

Si bien ADF automatiza las tareas de identificación y validación, y trabaja con defectos a nivel de capa, componente y paquetes, *jSmellSensor* es un plugin para Eclipse, que automatiza las tareas de definición, experimentación, identificación y validación del proceso descrito en la Sec.4.1. El diseño del prototipo utiliza el framework presentado en la Sec. 5.3. Los defectos que se gestionan son defectos a nivel de clase o método.

La integración con Eclipse se ha basado en la creación de un menú que da acceso a la funcionalidad mediante ventanas de diálogo y dos nuevas vistas de Eclipse utilizadas para presentar las tablas de defectos-entidades (ver Sec.4.1).

El plugin *jSmellSensor* utiliza las medidas de código calculadas con el plugin de Eclipse *RefactorIT* [Aqr02]. Desde *RefactorIT* se deben exportar las medidas en un fichero en formato xml. Este es el fichero que sirve como entrada al prototipo para poder llevar a cabo las tareas de identificación y experimentación. Además, al igual que ADF, integra cierta funcionalidad de la herramienta de minería de datos Weka [Wai07], en concreto, la implementación del algoritmo de clasificación J48, la representación gráfica del árbol de clasificación que se genera y las medidas de eficiencia de los clasificadores generados (recuperación, precisión, *F-Measure*) necesarias para validarlos.

El plugin *jSmellSensor* permite realizar las siguientes operaciones, organizadas según las tareas del proceso de detección de defectos propuesto:

- Tarea de Definición
 - Definir nuevos defectos de código desde cero.
 - Realimentar los clasificadores de los defectos ya definidos con nuevas instancias de entidades de código con o sin defectos.
 - Mostrar y guardar información sobre los defectos de código: descripción, reglas de detección y criterios de validación.
- Tarea de Experimentación
 - Crear clasificadores con entidades de código, sus medidas y la información sobre naturaleza de diseño (estereotipos UML).
 - Definir una clasificación de entidades en función de sus nombres cualificados. En nuestro caso una clasificación basada en los estereotipos UML: *Exception*, *Interface*, *Entity*, *Control*, *Test*, *Utility*.
- Tarea de Identificación
 - Detectar defectos de código en entidades.
 - Visualizar y modificar las predicciones iniciales de defectos en las entidades.
- Tarea de Validación
 - Comparar distintos clasificadores.

En las Fig. 5.11, 5.12, 5.13, 5.14, 5.15 y 5.16 se muestran las interfaces gráficas de usuario para interactuar con dichas tareas.

La tarea de identificación tiene como productos de entrada el profile con las definiciones de defectos de diseño a identificar y las entidades que se quieren evaluar. Produce como salida los resultados de la identificación en forma de tabla defectos-entidades. En el panel superior izquierdo de la Fig. 5.11, se muestra el profile de defectos de diseño disponible en este momento en el plugin (*Data Class*, *Feature Envy*, *God Class* y *Long Method*). Al tener seleccionado el defecto *Data Class* se visualizan en los distintos paneles los elementos que componen la definición del defecto: descripción y clasificadores asociados. En el panel inferior, por cada clasificador asociado al defecto, se muestra el árbol de decisión y los criterios de validación. Por último, se necesita especificar el fichero xml, obtenido con *RefactorIT*, que contiene las entidades de código sobre las que se quiere inspeccionar si tienen defectos y sus medidas.

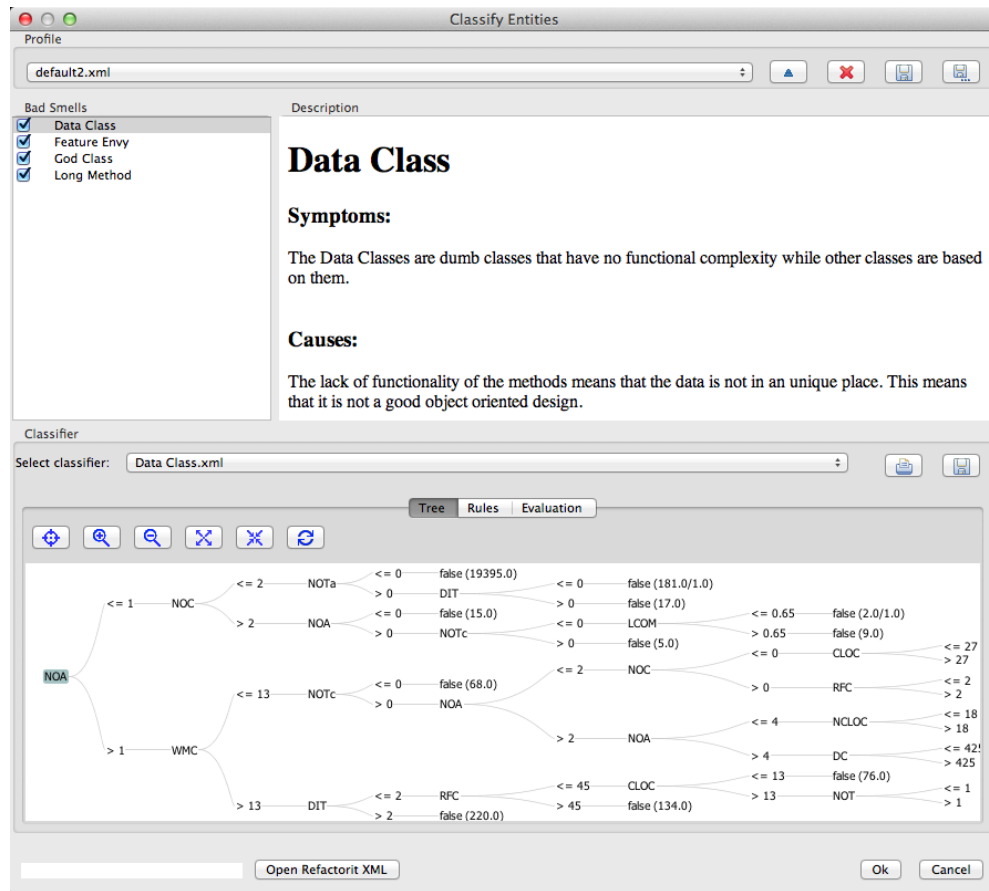
En la Fig. 5.12 se muestra la salida de la tarea de identificación. Este producto fue denominado en la Sec. 4.1 tabla inicial de defectos-entidades. Además, se permite al usuario modificar los distintos valores de las celdas para generar la tabla revisada de defectos-entidades.

Los defectos que conforman el profile deben ser previamente definidos por el usuario. En la Fig. 5.13 se muestra el diálogo para gestionar la definición de defectos de diseño. El panel izquierdo permite añadir, modificar o eliminar un defecto. Cuando se selecciona un defecto el panel derecho contiene sus elementos descriptivos. El campo descriptivo ARFF file hace referencia al formato de fichero de Weka que contiene las entidades iniciales clasificadas con o sin defecto. Esta es la entrada necesaria para obtener el árbol de decisión.

La tarea de experimentación tiene como objetivo generar nuevos clasificadores. La generación puede tener varias estrategias no excluyentes: modificar el conjunto de entidades o cambiar ciertas propiedades utilizadas para obtener el clasificador (validación cruzada y costes de falsos negativos que se ha observado como significativo en la experimentación del Cap. 4). En las Fig. 5.14 y 5.15 se muestran dos diálogos para interactuar con la tarea de experimentación. El diálogo de la Fig. 5.14 sirve para generar nuevos clasificadores, activando la casilla de comprobación etiquetada como *feedback*, se añaden las instancias de la tabla defectos-entidades revisada (ver Sec.4.1), para generar, mediante el algoritmo J48 un nuevo clasificador con esos datos. El diálogo de la Fig. 5.15 sirve para definir las entradas del algoritmo de clasificación según la naturaleza de diseño de la entidad basado en convención de nombres de la entidad (ver Alg. 1). El panel izquierdo contiene categorías de la naturaleza de diseño. Cuando se selecciona una categoría, se asocian las cadenas de texto que debe tener el nombre de una entidad para ser clasificada en la categoría seleccionada. La representación interna de esta información se hace mediante ficheros de configuración, con el formato de cuádruplas especificado en Sec. 5.2.1.2.

Respecto a la tarea de validación, cuando se genera un nuevo clasificador se dispone de un conjunto de indicadores para validar la eficiencia del nuevo clasificador respecto del anterior. En la Fig. 5.16 se muestran dos diálogos con los resultados de dos compara-

Figura 5.11. Interfaz gráfica del prototipo para seleccionar el profile en la tarea de identificación de defectos de diseño



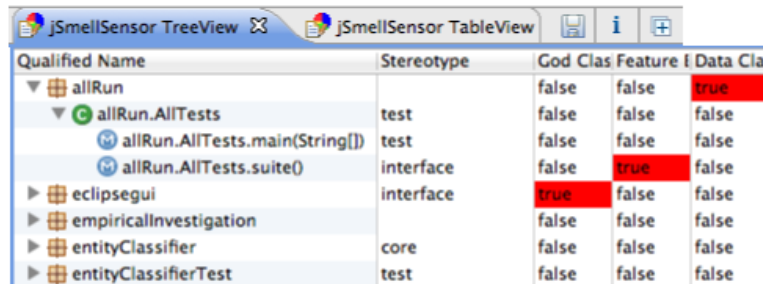
ciones de clasificadores. En uno se muestra la vista para comparar respecto al árbol de decisión. En el otro, se muestra la vista para comparar los clasificadores respecto a sus indicadores de eficiencia.

5.4.3. Conclusiones sobre el prototipado de gestión de defectos

Los prototipos de la gestión de defectos presentados proporcionan una prueba de concepto y de transferencia de conocimientos de nuestra propuesta de proceso gestión de defectos, presentada en Sec. 4.1. Han servido para validar el framework definido en la Sec. 5.3 y para validar el diseño de una interfaz gráfica que permita al usuario interactuar con las tareas del proceso. Con esto se consigue el objetivo para el que fueron definidos estos prototipos.

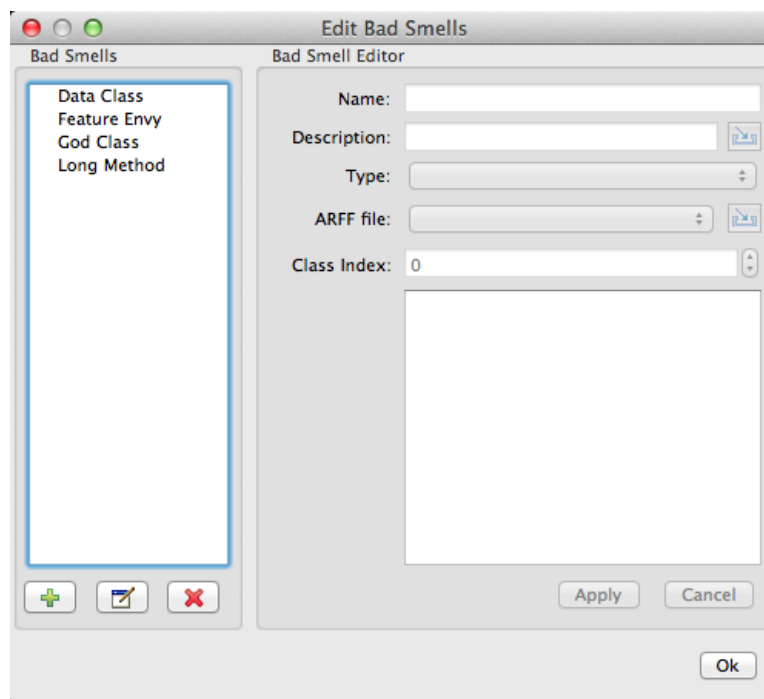
La utilización del framework de la Sec. 5.3, lleva asociado todas las ventajas e inconvenientes enumerados en la Sec. 5.3.3. Se destaca, por no tratarse en propuestas de otros autores, la inclusión de la información de la naturaleza de diseño de la entidad en el proceso de gestión de defectos.

Figura 5.12. Interfaz gráfica del prototipo con las tablas de entidades-defectos inicial y revisada



Qualified Name	Stereotype	God Clas	Feature	Data Cla
allRun		false	false	true
allRun.AllTests	test	false	false	false
allRun.AllTests.main(String[])	test	false	false	false
allRun.AllTests.suite()	interface	false	true	false
eclipsegui	interface	true	false	false
empiricalInvestigation		false	false	false
entityClassifier	core	false	false	false
entityClassifierTest	test	false	false	false

Figura 5.13. Interfaz gráfica del prototipo para la tarea de definición de defectos de diseño



The 'Edit Bad Smells' dialog box is divided into two main sections: 'Bad Smells' and 'Bad Smell Editor'.

Bad Smells: A list box containing the following items: Data Class, Feature Envy, God Class, and Long Method. Below the list are three buttons: a green plus sign (+), a pencil icon, and a red minus sign (-).

Bad Smell Editor: A form with the following fields and controls:

- Name:** A text input field.
- Description:** A text input field with a small icon to its right.
- Type:** A dropdown menu.
- ARFF file:** A dropdown menu with a small icon to its right.
- Class Index:** A text input field containing the value '0' and a spinner control to its right.

At the bottom of the dialog are three buttons: 'Apply', 'Cancel', and 'Ok'.

Desde el punto de vista de los autores, nos surgen dudas si sería necesario tener todas las actividades del proceso de gestión integradas en una única herramienta. La razón fundamental es la exigencia de conocimientos teóricos del usuario relacionados con la Ingeniería del Software y de Minería de Datos. En caso de disponer en la organización personal con ambos perfiles profesionales, de Ingeniero del Software y de Minería de Datos, se aconseja trabajar con la versión que automatiza todas las tareas, denominada gestión integral. Si no se dispone de personal con esta cualificación se recomendaría trabajar con el otro enfoque no integral.

Figura 5.14. Interfaz gráfica del prototipo para la tarea de experimentación: generar nuevos clasificadores

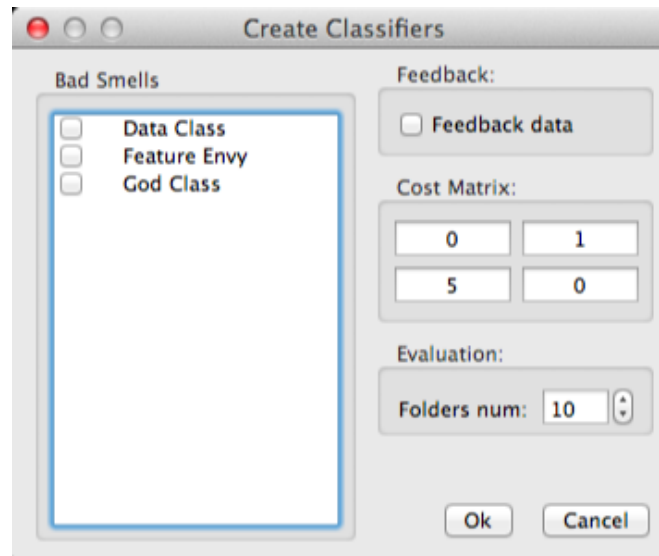


Figura 5.15. Interfaz gráfica del prototipo para la tarea de experimentación: configurar clasificación de la naturaleza de la entidad basada en nombres

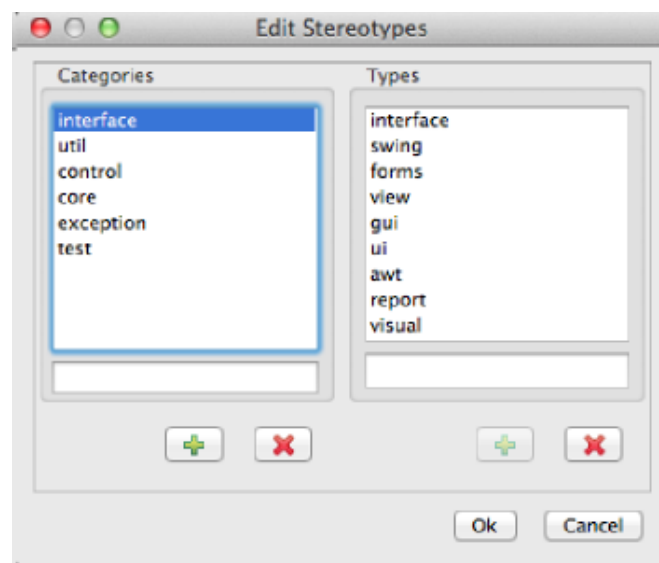
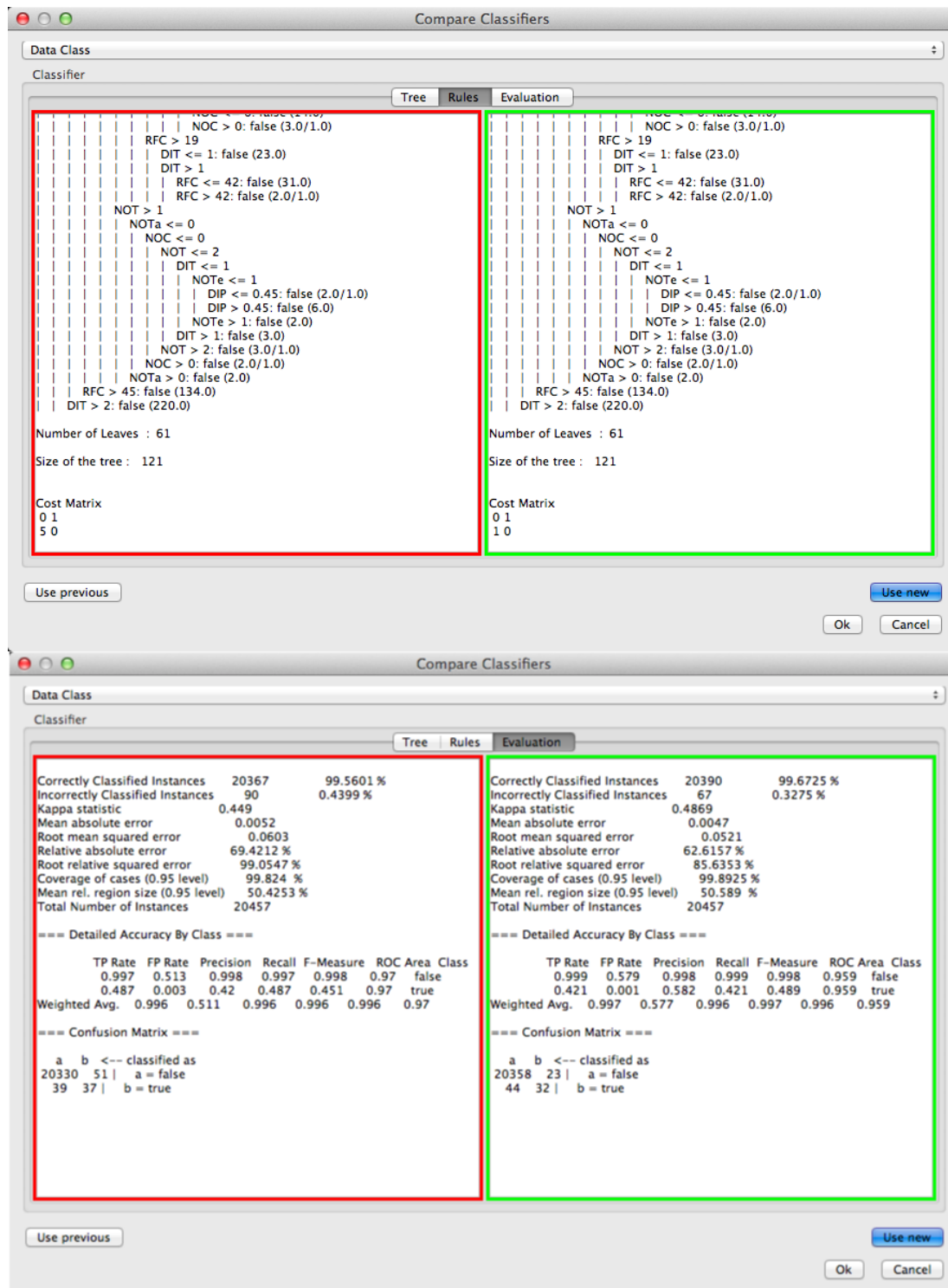


Figura 5.16. Interfaz gráfica del prototipo para la tarea de validación de los clasificadores



CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURAS

El objetivo principal de esta tesis es *desarrollar nuevos métodos y técnicas para mejorar la interpretación de las medidas de métricas de código, aplicadas a la investigación en la detección de defectos de diseño en orientación a objetos.*

Este objetivo se considera de interés porque puede mejorar la fase de mantenimiento de sistemas software orientados a objetos. En concreto, soporta la evaluación de la calidad en base a defectos de diseño, estableciendo un puente entre definiciones textuales y definiciones más precisas basadas en árboles de decisión, a partir de medidas de código.

En relación con la consecución de este objetivo, en este capítulo se presentan las conclusiones de esta tesis y las líneas de trabajo futuras. La presentación de las conclusiones se estructura por capítulos siguiendo los objetivos parciales especificados en la Sec.1.5.

6.1. Conclusiones

El segundo capítulo contiene un estudio detallado y estructurado de las diferentes propuestas en la gestión de defectos de diseño en orientación a objetos, usando la técnica de modelado de características. Así se cubren los objetivos parciales siguientes:

- Estado del arte de las herramientas que miden código.
- Estado del arte de las herramientas que gestionan defectos de diseño.

En conclusión, este capítulo aporta a los investigadores, desarrolladores o proveedores de herramientas las siguientes informaciones:

1. Soluciones concretas dentro de este dominio.

2. Un marco de trabajo para comparar y combinar diferentes herramientas individuales y enfoques.
3. Identificación y evaluación de herramientas para especificar una actividad de la gestión de defectos.
4. Un resumen de la investigación en el campo de gestión de defectos de diseño.
5. Identificación de problemas pendientes por resolver y desafíos que sugieran nuevas vías de investigación.

En el tercer capítulo se propone un nuevo proceso de medición, incorporando una categorización de las entidades de código, que mejora la interpretación los valores umbrales de sus métricas en función de las categorías. La confirmación de las mejoras de este proceso se ha llevado por medio de experimentación empírica. Los objetivos abordados en este capítulo y planteados en la Sec.1.5 son:

- Seleccionar un conjunto de métricas obteniendo valores umbrales para identificar medidas anómalas sin considerar la naturaleza de la entidad a medir.
- Aplicar un nuevo proceso de medición sobre las métricas elegidas considerando la naturaleza de la entidad y obtener unos valores umbrales para identificar medidas anómalas.
- Comprobar de manera empírica mediante experimentación, que la interpretación de las métricas aplicando los nuevos valores umbrales, mejora la precisión en la identificación de medidas anómalas.

Como conclusión, se presentan las siguientes observaciones de los estudios empíricos presentados en este capítulo:

1. Los valores umbrales de las medidas son diferentes entre los diferentes productos.
2. Los valores umbrales se mantienen a partir de versiones estables de productos.
3. Los valores umbrales de las medidas son diferentes para cada estereotipo UML.
4. La clasificación de entidades en estereotipos UML obtenida con el algoritmo propuesto concuerda con la clasificación que puede proporcionar un experto humano.
5. La concordancia de las clasificaciones de entidades de código en estereotipos UML, hechas por expertos aumenta cuando los sujetos tienen experiencia.

En el cuarto capítulo se define un proceso genérico y dinámico de gestión de defectos basado en métricas de código, en la naturaleza o intención de diseño de la entidad y en clasificadores binarios. La intención de diseño de la entidad sirve para modelar casos excepcionales documentados en la definición de defectos :*Data Class* vs. clases entity (Hibernate), *Feature Envy* vs. métodos test. Ninguna de las propuestas estudiadas previamente tratan los casos excepcionales en la identificación de los defectos. Las características diferenciadoras de ésta, frente a otras propuestas, son:

1. Retroalimentar la definición de un defecto para adaptarlo a un contexto de aplicación. El analista o auditor pueda personalizar sus reglas de detección de ma-

nera automática añadiendo nuevas entidades con defecto, o eliminando algunas, o etiquetando alguna con un nuevo estereotipo UML. La tarea es similar a la personalización de reglas de SPAM en un servicio de correo electrónico.

2. El proceso incorpora una tarea que sirve para validar las reglas de detección en un contexto de aplicación. La validación se hace con un indicador que relaciona la precisión y recuperación de resultados.

Como caso de estudio se aplica ese proceso a tres defectos de diseño: *God Class*, *Data Class* y *Feature Envy*. Se utiliza la experimentación empírica, con el objetivo de poder observar, si tener en cuenta los estereotipos UML, mejora los métodos de identificación de defectos de diseño.

Los objetivos abordados en este capítulo y planteados en la Sec.1.5 son:

- Definir un proceso de detección de defectos usando valores umbrales dependientes de la naturaleza de la entidad, y considerando la subjetividad del auditor.
- Seleccionar un conjunto de defectos y proporcionar clasificadores binarios (basados en reglas o árboles de decisión) a partir de métricas de código y la naturaleza de la entidad medida.
- Comprobar de manera empírica mediante experimentación los resultados de los clasificadores de detección de defectos frente a resultados de otras herramientas de detección.

Como conclusión se presentan las siguientes observaciones de los estudios empíricos presentados en este capítulo:

1. Las reglas de identificación de defectos varían en los distintos productos. Como consecuencia, la bondad de los resultados de la identificación, medida con un indicador basado en la recuperación y precisión, varía en función del producto.
2. Las reglas de identificación actuales difieren del método que se utilice. Cada método utiliza sus heurísticas propietarias definidas muchas veces a partir de medidas propietarias.
3. La información del estereotipo de la entidad mejora los resultados de la identificación.

En el quinto capítulo se aborda el diseño de frameworks y prototipos de herramientas que faciliten la transmisión tecnológica de las propuestas teóricas y experimentales presentadas en capítulos anteriores.

Los objetivos abordados en este capítulo y planteados en la Sec.1.5 son:

- Proponer un diseño genérico para un motor de cálculo de métricas.
- Definir un prototipo de herramienta software de cálculo métricas de código que incorpore la naturaleza de la entidad en el proceso de medición.
- Definir un prototipo de herramienta software de detección de defectos donde se incorporen las nuevas tareas del proceso del proceso de detección propuesto.

6.2. Líneas de Trabajo Futuras

Una de las piedras angulares de esta tesis, es incorporar tanto en los procesos de medición como en los de gestión de defectos de diseño, la naturaleza de diseño de la entidad, medida como estereotipos UML. Se ha definido un algoritmo que tiene como entrada el nombre cualificado de la entidad y una tabla de convención de nombres definida por el auditor que relaciona un conjunto de nombres con un estereotipo UML. A partir de las entradas se clasifica la entidad con un estereotipo UML o con una categoría especial de “unknown”. Debido a la propia estrategia de clasificación, los resultados obtenidos por el algoritmo tiene un escaso número de falsos positivos y un porcentaje más alto de falsos negativos. El algoritmo puede ser mejorado si recibe como información de entrada la propia entidad para poder ser analizada por introspección y conocer las dependencias con otras clases que ayuden a inferir el estereotipo.

El escenario más favorable para un algoritmo que clasifica entidades de código, es que la información del estereotipo esté incluida en la propia entidad de código en el momento de su creación, por ejemplo mediante anotaciones Java. Si no se cuenta con esto, el algoritmo ayuda a clasificar, pero los resultados siempre deberán estar supervisados por el auditor.

Con las mejoras del algoritmo, se pueden completar los resultados de los casos de estudio del Cap. 3, diseñado para evaluar la clasificación de entidades en estereotipos UML por expertos humanos y el algoritmo propuesto. La propuesta de mejora se basa en:

- Conseguir más información de evaluadores expertos.
- Clasificar otras entidades de código para contrastar la generalidad de los resultados.
- Estudiar posibles refinamientos de la clasificación.

A partir de las mejoras expuestas en el algoritmo se pueden definir más réplicas que permitan generalizar la hipótesis de que los valores umbrales de mediciones están condicionados por la naturaleza del problema que resuelve la entidad de código.

Por otra parte, en relación con el proceso de detección propuesto en el Cap. 4, sería interesante trabajar en las siguientes direcciones:

- Aplicar y evaluar los resultados de otros métodos de clasificación de caja blanca distintos de los árboles de decisión, como los basados en reglas.
- Aplicar otras técnicas para enfrentarse al problema del conjunto de datos desequilibrados. Proponer una solución específica de dominio, mediante la inyección artificial de entidades con defecto, generadas a partir de una teoría de conocimiento base o resultados de otros autores.
- Aplicar técnicas de clasificación que permitan obtener una clasificación difusa, esto es, una estimación de la probabilidad de que una entidad tenga un defecto.

En el estudio empírico expuesto en el Cap. 4 y definido para analizar la influencia de la naturaleza de la entidad en las reglas de identificación de defectos, creemos que se

necesitan más réplicas para validar la hipótesis. Las nuevas réplicas [Moo05] se pueden definir de la siguiente forma:

- Aumentando los conjuntos de datos, obteniendo más entidades con defectos de otros proyectos similares.
- Realizando la experimentación sobre otros defectos de diseño que pueden ser detectados con métricas.
- Añadiendo variables con medidas obtenidas de otras fuentes distintas del código. Por ejemplo, pueden utilizarse métricas de proceso relacionadas con el número de cambios que ha sufrido una entidad.
- Incorporando información sobre la clasificación de las entidades respecto de otros defectos. Esto permitiría establecer correlaciones entre los distintos defectos.

Por su parte, los prototipos presentados el Cap. 5 necesitan ser desarrollados completamente para poder pasar a su implementación en entornos productivos. Así se podría evaluar si cumplen sus objetivos finales, es decir, si realmente su uso mejora la actividad de mantenimiento del software.

Apéndices

ENCUESTA DE CLASIFICACIÓN DE ENTIDADES DE CÓDIGO

A.1. Descripción

El objetivo de esta encuesta es evaluar un algoritmo de clasificación de entidades de código orientadas a objetos según los siguientes estereotipos UML: exception, interface, entity, control, test, utility.

En la encuesta se pide que se clasifiquen varias entidades de código para poder comparar tu clasificación con la que resulta al ejecutar el algoritmo. El tiempo estimado para realizarla es de 30-45 minutos.

Este trabajo ha sido realizado dentro del grupo de investigación GIRO de la Universidad de Valladolid y está financiado por el Ministerio de Ciencia e Innovación a través del proyecto de investigación ROADMAP (TIN2008-05675). Encuesta disponible electrónicamente en <http://www.encuestafacil.com/RespWeb/Qn.aspx?EID=788250>

A.2. Perfil del inspector de código

- *1. Sexo
- *2. Edad
- *3. ¿Cuál es tu ocupación actual? (Estudiante, Profesor, Profesional)
- *4. ¿Qué experiencia tienes en programar aplicaciones en orientación a objetos?(0, 2, 5, entre 5 y 10, más de 10 años)
- *5. ¿Cuál es tu conocimiento del lenguaje de programación Java? (Nulo, Bajo, Medio, Avanzado, Muy avanzado)
- *6. ¿Cuál es tu conocimiento sobre UML? (Nulo, Bajo, Medio, Avanzado, Muy avanzado)

A.3. Estereotipos UML

Definiciones básicas de estereotipos UML sobre clases y paquetes.

*1. Selecciona la definición que corresponde con el estereotipo *Excepción*

- Representa un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
- Representa una entidad de código que actúa como interfaz de un sistema o subsistema.
- Representa un concepto del problema o del negocio.
- Representa un objeto intermediario que coordina y conecta la interacción de múltiples objetos generalmente de subsistemas distintos.
- Colección de funciones que crean clases y objetos para probar el funcionamiento del código.
- Colección de funciones estáticas. La clase no tiene instancias.

*2. Selecciona la definición que corresponde con el estereotipo *Entity*

- Representa un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
- Representa una entidad de código que actúa como interfaz de un sistema o subsistema.
- Representa un concepto del problema o del negocio.
- Representa un objeto intermediario que coordina y conecta la interacción de múltiples objetos generalmente de subsistemas distintos.
- Colección de funciones que crean clases y objetos para probar el funcionamiento del código.
- Colección de funciones estáticas. La clase no tiene instancias.

*3. Selecciona la definición que corresponde con el estereotipo *Interface*

- Representa un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
- Representa una entidad de código que actúa como interfaz de un sistema o subsistema.
- Representa un concepto del problema o del negocio.
- Representa un objeto intermediario que coordina y conecta la interacción de múltiples objetos generalmente de subsistemas distintos.

- Colección de funciones que crean clases y objetos para probar el funcionamiento del código.
 - Colección de funciones estáticas. La clase no tiene instancias.
- *4. Selecciona la definición que corresponde con el estereotipo *Test*
- Representa un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
 - Representa una entidad de código que actúa como interfaz de un sistema o subsistema.
 - Representa un concepto del problema o del negocio.
 - Representa un objeto intermediario que coordina y conecta la interacción de múltiples objetos generalmente de subsistemas distintos.
 - Colección de funciones que crean clases y objetos para probar el funcionamiento del código.
 - Colección de funciones estáticas. La clase no tiene instancias.
- *5. Selecciona la definición que corresponde con el estereotipo *Utility*
- Representa un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
 - Representa una entidad de código que actúa como interfaz de un sistema o subsistema.
 - Representa un concepto del problema o del negocio.
 - Representa un objeto intermediario que coordina y conecta la interacción de múltiples objetos generalmente de subsistemas distintos.
 - Colección de funciones que crean clases y objetos para probar el funcionamiento del código.
 - Colección de funciones estáticas. La clase no tiene instancias.
- *6. Selecciona la definición que corresponde con el estereotipo *Control*
- Representa un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
 - Representa una entidad de código que actúa como interfaz de un sistema o subsistema.
 - Representa un concepto del problema o del negocio.
 - Representa un objeto intermediario que coordina y conecta la interacción de múltiples objetos generalmente de subsistemas distintos.
 - Colección de funciones que crean clases y objetos para probar el funcionamiento del código.

- Colección de funciones estáticas. La clase no tiene instancias.

A.4. Clasificación de entidades de código de la aplicación Visor de Imágenes

En esta página se proponen varias entidades de código (clases y paquetes) de una pequeña aplicación Java para que las clasifiques según su naturaleza. Se entiende por naturaleza los siguientes estereotipos estándar de UML: e1 exception, e2 interface, e3 entity, e4 control, e5 test, e6 utility. Además se considera una opción especial “desconocida” para indicar que no se sabe clasificar esa entidad. Para poder clasificar las entidades propuestas inicialmente se utilizará sólo información de la arquitectura del sistema y después volverás a clasificar las mismas entidades utilizando código fuente.

*1. La aplicación es un applet Java que permite visualizar un conjunto de imágenes cuyos datos (autor, descripción, nombre de fichero/url) se encuentran almacenados en un sistema de almacenamiento persistente (fotografías). El acceso a los datos del sistema de almacenamiento persistente se realiza a través de una aplicación que reside en el lado del servidor web. En la Fig. A.1 se muestra un diseño de la interfaz de usuario para interactuar con los datos. ¿Entiendes la funcionalidad de la aplicación? (Si, No)

Figura A.1. Interfaz gráfica visor de imágenes



*2. En la Fig. A.2 se muestra el conjunto de clases que conforman la aplicación y su organización en paquetes. ¿Entiendes el criterio utilizado para agrupar las clases? (Si, No)

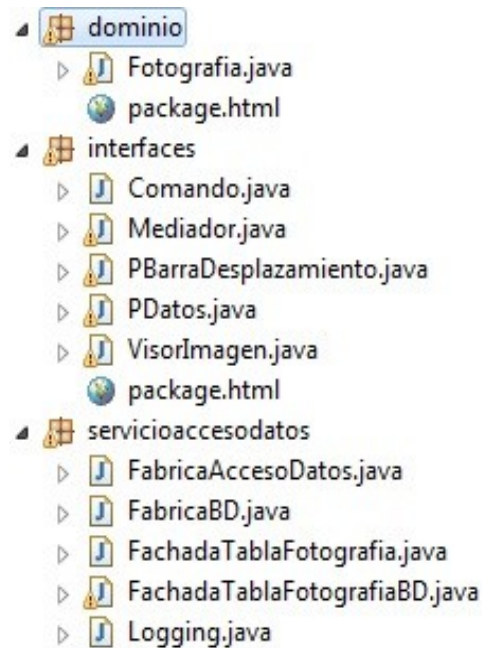
*3. ¿En que categoría clasificas la clase dominio.Fotografía? (exception, interface, entity, control, test, utility, desconocida)

*4. ¿En qué categoría clasificas la clase interfaces.VisorImagenes ? (exception, interface, entity, control, test, utility, desconocida)

*5. ¿ En que categoría clasificas la clase servicioaccesodatos.FachadaTablaFotografíaBD? (exception, interface, entity, control, test, utility, desconocida)

*6. ¿En qué categoría clasificas la clase servicioaccesodatos.Logging? (exception,

Figura A.2. Estructura y nombres de las entidades de código



interface, entity, control, test, utility, desconocida)

*7. Dado el código fuente de la clase dominio.Fotografía disponible en <http://xp-dev.com/sc/browse/75606/%2Fsrc%2Fmodel%2FPhoto.java?rev=-2> ¿En qué categoría la clasificas? (exception, interface, entity, control, test, utility, desconocida)

*8. Dado el código fuente de la clase interfaces.VisorImagen disponible en <http://xp-dev.com/sc/browse/75606/%2Fsrc%2Finterfaces%2FImageViewer.java?rev=-2> ¿En qué categoría la clasificas? (exception, interface, entity, control, test, utility, desconocida)

*9. Dado el código fuente de la clase servicioaccesodatos.FachadaFotografíaBD disponible en <http://xp-dev.com/sc/browse/75606/%2Fsrc%2Fdatapersistentaccess%2FFacadePhotoTableDB.java?rev=-2> ¿En qué categoría la clasificas? (exception, interface, entity, control, test, utility, desconocida)

*10. Dado el código fuente de la clase servicioaccesodatos.Logging disponible en <http://xp-dev.com/sc/browse/75606/%2Fsrc%2Fdatapersistentaccess%2FLogging.java?rev=-2> ¿En qué categoría la clasificas? (exception, interface, entity, control, test, utility, desconocida)

2FLogging.java?rev=-2 ¿En qué categoría la clasificas? (exception, interface, entity, control, test, utility, desconocida)

A.5. Clasificación de entidades de código EclEmma a partir de código fuente

En esta página se proponen varias entidades de código (clases y paquetes) de un sistema de código abierto (EclEmma <http://www.eclemma.org/>) para que las clasifiques según su naturaleza. Se entiende por naturaleza los siguientes estereotipos estándar de UML: e1 exception, e2 interface, e3 entity, e4 control, e5 test, e6 utility. Además se considera una opción especial “desconocida” para indicar que no se sabe clasificar esa entidad. Para poder clasificar la entidad te puedes basar en consultar el código fuente de la entidad o consultar información del contexto donde se encuentra, por ejemplo el subsistema en el que se encuentra o su documentación.

*1. A partir de la funcionalidad de la aplicación EclEmma descrita en <http://www.eclemma.org/>. ¿Entiendes el dominio de la aplicación y su funcionalidad externa? (Si, No)

*2. ¿En tus desarrollos has trabajado alguna vez con herramientas de cobertura de pruebas? (Si, No)

*3. ¿En qué categoría clasificas la clase disponible en <http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/com.mountainminds.eclemma.ui/src/com/mountainminds/eclemma/internal/ui/UIPreferences.java?revision=1142&view=markup> ? (exception, interface, entity, control, test, utility, desconocida)

*4. ¿En qué categoría clasificas la clase disponible en <http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/com.mountainminds.eclemma.ui/src/com/mountainminds/eclemma/internal/ui/dialogs/MergeSessionsDialog.java?revision=1142&view=markup> ?(exception, interface, entity, control, test, utility, desconocida)

*5. ¿En qué categoría clasificas la clase disponible en <http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/com.mountainminds.eclemma.ui/src/com/mountainminds/eclemma/internal/ui/actions/MergeSessionsAction.java?revision=1142&view=markup> ? (exception, interface, entity, control, test, utility, desconocida)

*6. ¿En qué categoría clasificas la clase disponible en <http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/com>.

`mountainminds.eclemma.core/src/com/mountainminds/eclemma/internal/core/SessionManager.java?revision=1142&view=markup` ? (exception, interface, entity, control, test, utility, desconocida)

*7. ¿En qué categoría clasificas la clase disponible en `http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/com.mountainminds.eclemma.core/src/com/mountainminds/eclemma/internal/core/analysis/JavaModelCoverage.java?revision=1142&view=markup` ? (exception, interface, entity, control, test, utility, desconocida)

*8. ¿En qué categoría clasificas la clase disponible en `http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/com.mountainminds.eclemma.core/src/com/mountainminds/eclemma/internal/core/analysis/JavaElementCoverage.java?revision=1142&view=markup` ? (exception, interface, entity, control, test, utility, desconocida)

*9. ¿En qué categoría clasificas la clase disponible en `http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/com.mountainminds.eclemma.core.test/src/com/mountainminds/eclemma/internal/core/SessionManagerTest.java?revision=1142&view=markup` ? (exception, interface, entity, control, test, utility, desconocida)

*10. ¿En qué categoría clasificas la clase disponible en `http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/com.mountainminds.eclemma.ui/src/com/mountainminds/eclemma/internal/ui/coverageview/CoverageView.java?revision=1142&view=markup` ? (exception, interface, entity, control, test, utility, desconocida)

*11. ¿En qué categoría clasificas las clases del subsistema `com.mountainminds.eclemma.ui` disponible en `http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/` ? (exception, interface, entity, control, test, utility, desconocida)

*12. ¿En qué categoría clasificas las clases del subsistema `com.mountainminds.eclemma.core.test` disponible en `http://eclemma.svn.sourceforge.net/viewvc/eclemma/eclemma/tags/v1.5.1/` ? (exception, interface, entity, control, test, utility, desconocida)

13. ¿Puedes indicar la url de una entidad de esta aplicación que clasifiques con el estereotipo exception?

14. ¿Puedes indicar la url de una entidad de esta aplicación que clasifiques con el

estereotipo interface?

15. ¿Puedes indicar la url de una entidad de esta aplicación que clasifiques con el estereotipo entity?

16. ¿Puedes indicar la url de una entidad de esta aplicación que clasifiques con el estereotipo control?

17. ¿Puedes indicar la url de una entidad de esta aplicación que clasifiques con el estereotipo test?

18. ¿Puedes indicar la url de una entidad de esta aplicación que clasifiques con el estereotipo utility?

19. ¿Puedes indicar la url de una entidad de esta aplicación que no puedas clasificar con los estereotipos proporcionados (categoría desconocida)?

PUBLICACIONES

Esta tesis surge para enfrentarse parcialmente a los objetivos generales de investigación del grupo de la Universidad de Valladolid GIRO (Grupo de Investigación en Reutilización y Orientación a objetos) centrados en el campo de evolución y mantenimiento del software mediante refactorizaciones.

En las etapas iniciales de la investigación de esta tesis se creo un grupo de trabajo de cuatro miembros, tres doctorandos, entre los que se incluye el autor, y como líder la Dra. Yania Crespo. Las publicaciones de estas etapas estaban centradas en los objetivos generales del grupo basados en definir un entorno de mantenimiento con independencia del lenguaje de programación. Posteriormente se establecieron tres temas de trabajo independientes: detección de defectos, corrección y definición de refactorizaciones. Cada área fue abordada por un miembro dando lugar a publicaciones relacionadas directamente con capítulos de esta tesis.

Todos los capítulos de la tesis han sido tratados, por lo menos parcialmente, en alguna publicación de taller o congreso. Estas publicaciones han servido por una lado, para validar las propuestas y por otro lado, para mejorar las propuestas con los comentarios de sus revisores.

En este anexo se indican los congresos, talleres y revistas donde se han realizado alguna publicación. Además se presenta una descripción breve de las publicaciones y su relación con los diferentes capítulos de esta tesis.

B.1. Congresos, talleres y revistas

La participación en los talleres especializados ha ayudado a concretar los objetivos de la tesis y a obtener un conjunto de trabajos relacionados. Los talleres en los que se ha participado han sido:

- QAOOSE (Workshop on Quantitative Approaches in Object-Oriented Software Engineering), es un taller especializado en medición de sistemas orientados a

objetos. Se ha participado con publicación en tres ediciones del taller: novena (2005), décima (2006) y décimo tercera (2010). Este taller ha estado integrado dentro de dos congresos de relevancia internacional: ECOOP (European Conference on Object-Oriented Programming), TOOLS (48th International Conference on Objects, Models, Components, Patterns).

- WOOR (workshop on Object-Oriented Reengineering), es un taller especializado en reingeniería y evolución del software. Se ha participado con publicación en dos ediciones del taller: sexta (2005) y séptima (2006). El taller estaba integrado dentro del congreso internacional ECOOP (European Conference on Object-Oriented Programming).

Los congresos en los que se ha participado están centrados en el contexto de la Ingeniería del Software y han servido para poder validar nuestras propuestas desde un punto de vista externo. En concreto se ha participado en los siguientes congresos:

- JISBD (Jornadas de Ingeniería del Software y Bases de Datos), es un congreso de perfil genérico sobre la Ingeniería del Software y Bases de Datos, está organizado en sesiones temáticas. En relación con las publicaciones presentadas se destaca la sesión temática denominada “Apoyo a la decisión en Ingeniería del Software, Metodologías, Experimentación”. Desde 2003 se ha participado en siete ediciones de este congreso con presentación de artículos.
- ICSOFT (International Conference on Software and Data Technologies), el congreso está organizado en áreas y se ha participado en la primera edición (2006) en el área de Ingeniería del Software.

Además se ha participado con dos artículos en la Revista de Procesos y Métricas (RPM) gestionada por AEMES (Asociación Española de Métricas de Sistemas Informáticos). La Revista de Procesos y Métricas (RPM) tiene dos objetivos fundamentales: El primero, convertirse en una publicación científica reconocida en el área de la gestión de los Procesos de las Tecnologías de la Información y Comunicaciones y en particular en todos los aspectos relacionados con el Software; y el segundo, servir de vehículo de comunicación entre el mundo académico y el empresarial. Se ha participado con dos artículos en ediciones del 2010 y 2011.

Una visión integradora de los objetivos de investigación generales de GIRO fue publicada como capítulo del libro titulado *Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices* [CLMM06b].

Además de las publicaciones aceptadas ha habido varios artículos rechazados en distintos congresos y revistas: JISBD, WCRE (Working Conference on Reverse Engineering), EMSE (Empirical Software Engineering), Journal of Software Maintenance and Evolution: Research and Practice.

B.2. Publicaciones y su relación con capítulos

A continuación se detallan las publicaciones ordenadas cronológicamente en relación con los capítulos de esta tesis. Las publicaciones están disponibles en el repositorio del grupo de investigación GIRO (ver <http://www.giro.infor.uva.es/Publications/index.php?idAuthor=11>). En la Tabla B.1 se recoge un cuadro resumen de las referencias cruzadas entre las distintas publicaciones realizadas y los capítulos, como resumen final.

En relación con los capítulos 1 y 5, como base del trabajo de investigación en [LC03], se abordó el estudio de la independencia del lenguaje y conceptos de orientación a objetos (OO).

Posteriormente y en relación con esos mismos capítulos se abordó una aproximación global para incorporar esta característica de independencia del lenguaje en un proceso de mantenimiento del software basado en refactorizaciones. En este sentido se participó en varias publicaciones compartidas con diferentes miembros del grupo de investigación GIRO. En concreto, en [LMC03], se recogió una primera propuesta de la plantilla de refactorizaciones con independencia del lenguaje, y una primera aproximación al uso de *frameworks*. En [MLC03] se profundizó sobre el primer desarrollo de un catálogo de refactorizaciones en especialización, en cuanto a genericidad. En [CLM04] se extendió el anterior trabajo, más concretamente en relación a la ejecución de las refactorizaciones, dejando una sólida base para su ejecución. En [LMC04] se presentó un póster con la visión global, en base a los anteriores trabajos.

En [MCL05] y [CLMM05], en relación con los capítulos 3, 4 y 5, se estudió cómo consultar la información almacenada con el fin de detectar defectos de código, a través de métricas y heurísticas. Se prosiguió con el uso de *frameworks* para la obtención de métricas con independencia del lenguaje, a través del diseño de un *framework* que permite obtener esta información. En esta línea se constató la posibilidad de obtener métricas independientes del lenguaje sobre plataformas como Java y .NET. Siguiendo en esta línea, en [MLC05a] se presentó el uso de métricas y heurísticas independientes del lenguaje (consultas) para sugerir la detección de defectos de código y poder aplicar el conjunto de refactorizaciones.

En [CLMM06b] se participó mediante el desarrollo de un capítulo del libro, con una revisión de los capítulos 2, 3, 4 y 5.

En [CLM06] y [MLC06a] se prosiguió con la clasificación y mecanismos para la detección de defectos de código en relación con los capítulos 2 y 4.

En relación con los capítulos 2 y 5, en [LMCP06] se presentó una comparativa entre el soporte de MOON y sus instanciaciones concretas, con sus ventajas y desventajas, frente al uso de UML y el uso de acciones semánticas. Dado el nivel de complejidad del metamodelo UML para representar código, se mostraron las ventajas del uso de una representación minimal como MOON. En [LMC06] se presentó una caracterización de refactorizaciones para su posible implementación en herramientas.

En [MLC07] se planteó el proceso a seguir de cara a la definición, construcción e implementación de refactorizaciones, analizando los pros y contras de distintas variantes,

y planteando una solución final de cara a consolidar todos los trabajos anteriormente expuestos. El trabajo se encuadra dentro del capítulo 2.

En relación con el capítulo 3, en [LCMM09] se define un caso de estudio donde el proceso de medición es guiado por el inspector/evaluador, que clasifica las entidades de código a medir según su naturaleza. Como resultado del caso de estudio se proponen unos valores umbrales de las métricas según la naturaleza de la entidad, y relativos a una organización concreta. Además se observa de manera empírica que los valores de las métricas de código dependen de la naturaleza de la entidad de código a medir.

En relación con los capítulo 2 y 5, en [LMC10] se identifican los requisitos que debe cumplir una herramienta de soporte a la medición de código para poder abordar la interpretación de métricas considerando la naturaleza de las entidades. En base a estos requisitos, se revisan algunas herramientas existentes y se constata la dificultad de aplicar esta propuesta con las funcionalidades actuales. Por ello se presenta la adaptación de una de las herramientas revisadas (RefactorIt) y, además, aplicando el proceso de medición sobre diez proyectos reales se obtienen unos valores umbrales iniciales condicionados por la naturaleza de las entidades de código.

En relación con el capítulo 2, en el informe técnico [PLMM11] se realiza una revisión del estado del arte de la gestión de defectos de diseño y se presenta un estudio exhaustivo de los enfoques y herramientas de defectos de diseño existentes. Mediante el uso de diagramas de características se ilustra de forma gráfica el estudio y se define una taxonomía.

En [LMC11] y en relación con capítulo 3, se presenta una réplica del caso de estudio presentado [LCMM09]. La réplica se crea modificando el conjunto de métricas, como resultado se vuelve a confirmar la existencia de una relación entre la naturaleza de las entidades y los valores de sus medidas.

En [LCMM11] y en relación con los capítulos 4 y 5, se define un proceso gestión de defectos de diseño. En la actividad de detección se considera la naturaleza de la entidades y se aplican técnicas de clasificación de minería de datos. Como prueba de concepto se presenta un plugin de Eclipse que automatiza parcialmente el proceso definido para un conjunto de defectos arquitectónicos.

En [LMC12] y en relación con el capítulo 4, se presenta el estudio empírico para evaluar la eficiencia de los métodos de detección del defecto God Class. Como conclusión del trabajo se observa que la naturaleza de la entidad, indicada a través de estereotipos estándar de clasificadores UML, influye en la identificación de entidades con el defecto God Class.

Tabla B.1. Referencias cruzadas entre publicaciones y capítulos

Capítulos	Referencias bibliográficas
C1	[LC03],[LMC03], [MLC03], [LMC04], [CLM04], [MLC07],[PLMM11]
C2	[CLMM06b],[MLC06a], [CLM06], [LMC10],[PLMM11]
C3	[MCL05], [CLMM05], [CLMM06b], [LCMM09], [LMC10], [LMC11]
C4	[MCL05], [CLMM05],[CLMM06b], [CLM06], [MLC06a], [LCMM11], [LMC12]
C5	[CLMM06b], [CLM06], [MLC06a], [LMC10], [LCMM11]

REFERENCIAS

- [ABF04] Erik Arisholm, Lionel C. Briand, and Audun Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Software Eng.*, 30(8):491–506, 2004.
- [ADN05] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 62–71, 2005.
- [AN05] J. Arlow and I. Neustadt. *Uml 2 And The Unified Process: Practical Object-oriented Analysis And Design*. Addison-Wesley Object Technology Series, 2005.
- [Ana08] Analyst4j. <http://www.codeswat.com>, February 2008. [Accessed: 2010-04-06].
- [Aqr02] Aqris. RefactorIT. <http://www.refactorit.com>, 2002. [Accessed: 2010-04-06].
- [Arg] ArgoUML. <http://argouml.tigris.org>. [Accessed: 2009-10-23].
- [AS04] Erik Arisholm and Dag I. K. Sjøberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Trans. Software Eng.*, 30(8):521–534, 2004.
- [AS09] El Hachemi Alikacem and Houari A. Sahraoui. A metric extraction framework based on a high-level description language. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 159–167, Washington, DC, USA, 2009. IEEE Computer Society.
- [BAMN06] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A novel approach to optimize clone refactoring activity. In *8th annual conference on Genetic and evolutionary computation*, pages 1885–1892, New York, NY, USA, 2006. ACM.

- [BBAA03] Pierre Bourque, Luigi Buglione, Alain Abran, and Alain April. Bloom's taxonomy levels for three software engineer profiles. In *STEP*, pages 123–129, 2003.
- [BBD01] Lionel C. Briand, Christian Bunse, and John W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Trans. Software Eng.*, 27(6):513–530, 2001.
- [BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, 1976.
- [BCR94] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach. The goal question metric approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley, 1994.
- [BCT07] Thierry Bodhuin, Gerardo Canfora, and Luigi Troiano. Sormasa: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. In *1st Workshop on Refactoring Tools [DC07]*, pages 23–24.
- [Bec97] Kent Beck. *Smalltalk: best practice patterns*. Prentice-Hall, Inc., 1997.
- [BEG⁺06] P. Baker, D. Evans, J. Grabowski, H. Neukirchen, and B. Zeiss. Trex - the refactoring and metrics tool for ttcn-3 test specifications. In *Testing: Academic and Industrial Conference - Practice And Research Techniques*, pages 90–94, Aug. 2006.
- [BEM95] L. Briand, K. El-Emam, and S. Morasca. Theoretical and empirical validation of software product measures. Technical report, International Software Engineering Research Network (ISERN-95-03), 1995.
- [BLYP04] Lionel C. Briand, Yvan Labiche, H.-D. Yan, and Massimiliano Di Penta. A controlled experiment on the impact of the object constraint language in uml-based development. In *ICSM*, pages 380–389, 2004.
- [BMB02] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. An operational process for goal-driven definition of measures, 2002.
- [BMMM98] 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.
- [Bor] Borland. Together. <http://www.borland.com/us/products/together>. [Accessed: 2010-04-06].
- [BSL99] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 25(4):456–473, 1999.
- [BVT03] Rajendra K. Bandi, Vijay K. Vaishnavi, and Daniel E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Trans. Software Eng.*, 29(1):77–87, 2003.

- [BWH98] L. Briand, J. Wüst, and H.Lounis. Replicated case studies for investigating quality factors in object-oriented designs. Technical report, International Software Engineering Research Network (ISERN-98-29-ver3), 1998.
- [CE00] Czarnecki and 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.
- [CHCK12] David Cieslak, T. Hoens, Nitesh Chawla, and W. Kegelmeyer. Hellinger distance decision trees are robust and skew-insensitive. *Data Mining and Knowledge Discovery*, 24(1):136–158, 2012. 10.1007/s10618-011-0222-1.
- [Che04] CheckStyle. <http://checkstyle.sourceforge.net>, 2004. [Accessed: 2010-04-06].
- [Chi02] Ciprian-Bogdan Chirila. Automation of the design flaw detection process in object-oriented systems. *International Conference on Technical Informatics (CONTI); Periodica Politechnica – Transactions on Automatic Control and Computer Science*, 47, October 2002.
- [Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *International Conference on Technology of Object-Oriented Languages and Systems*, pages 18–32, Washington, DC, USA, 1999. IEEE Computer Society.
- [CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *ACM SIGPLAN Notices*, 26(11):197–211, 1991.
- [CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493, 1994. Journal.
- [CLM04] Yania Crespo, Carlos López, and Raúl Marticorena. Un framework para la reutilizacion de la definicion de refactorizaciones. In Hernesto Pimentel Juan Hernandez, editor, *IX Jornadas Ingenieria del Software y Bases de Datos (JISBD 2004)*, Malaga, Spain ISBN: 84-688-8983-0, pages 499–506, nov 2004.
- [CLM06] Yania Crespo, Carlos López, and Raúl Marticorena. Relative thresholds: Case study to incorporate metrics in the detection of bad smells. In *10 th ECOOP Workshop on QAOOSE 06, Quantitative Approches in Object-Oriented Software Engineering*. Nantes, France. ISBN 88-6101-000-8. <http://www.inf.unisi.ch/faculty/lanza/QAOOSE2006/>, pages 109–118, 2006.
- [CLMM05] Yania Crespo, Carlos López, Raúl Marticorena, and Esperanza Manso. Language independent metrics support towards refactoring inference. In *9th ECOOP Workshop on QAOOSE 05 (Quantitative Approches*

- in Object-Oriented Software Engineering*). Glasgow, UK. ISBN: 2-89522-065-4, pages 18–29, jul 2005.
- [CLMM06a] Yania Crespo, Carlos López, Esperanza Manso, and Raúl Marticorena. From bad smells to refactoring, metrics smoothing the way. In *Object-Oriented Design Knowledge. Principles, Heuristics and Best Practices*, Object-Oriented Design Knowledge. Principles, Heuristics and Best Practices, chapter VII, pages 193–249. Idea Group Publishing, 2006.
- [CLMM06b] 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. ISBN: 1-50140-899-9. Idea Group Publishing, 2006. <http://www.amazon.com/Object-Oriented-Design-Knowledge-Principles-Heuristics/dp/1591408962>.
- [Con99] Clarkware Consulting. JDepend. <http://clarkware.com/software/JDepend.html>, 1999. [Accessed: 2010-04-06].
- [CPG01] Coral Calero, Mario Piattini, and Marcela Genero. Empirical validation of referential integrity metrics. *Information & Software Technology*, 43(15):949–957, 2001.
- [Cre00] Yania Crespo. *Incremento del potencial de reutilización del software mediante refactorizaciones*. PhD thesis, Universidad de Valladolid, 2000.
- [CS07] Campwood and Software. SourceMonitor, 2007.
- [DC07] Danny Dig and Michael Cebulla. 1st workshop on refactoring tools. Technical Report 2007-8, TU Berlin, July 2007.
- [DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002.
- [DDV⁺06] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering*, pages 346–355. IASTED/ACTA Press, 2006.
- [DeM82] T. DeMarco. *Controlling software projects*. Yourdon Press Prentice-Hall, 1982.
- [DF00] J.J. Dolado and L. Fernández. *Medición para la Gestión en la Ingeniería del Software*. Ra-Ma, 2000.
- [DLOV08] Andrea De Lucia, Rocco Oliveto, and Luigi Vorraro. Using structural and semantic metrics to improve class cohesion. In *International Conference on Software Maintenance*, pages 27–36. IEEE, October 2008.
- [Dro96] R. G. Dromey. Cornering the chimera. *IEEE Software*:33–43, 1996.
- [DSA⁺04] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an

- object oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, July 2004.
- [DSRS03] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139, 2003.
- [DTS99] 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.
- [Dur07] Nakul Durve. Coderaider: Automatically improving the design of code. diploma thesis, June 2007.
- [eAC94] F. Brito e Abreu and R. Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *J. Systems Software*, 26:87–96, 1994.
- [EBG⁺02] Khaled El Emam, Saïda Benlarbi, Nishith Goel, Walcélio L. Melo, Hakim Lounis, and Shesh N. Rai. The optimal class size for object-oriented software. *IEEE Trans. Software Eng.*, 28(5):494–509, 2002.
- [EBGR01] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, 2001.
- [Ecl] Eclipse Metrics Plugin. <http://eclipse-metrics.sourceforge.net>. [Accessed: 2010-04-06].
- [ED00] L. Etzkorn and H. Delugach. Towards a semantic metrics suite for object-oriented design. In *34th International Conference on Technology of Object-Oriented Languages and Systems*, pages 71–80, 2000.
- [Fag02] Michael Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers: contributions to software engineering*, pages 575–607. Springer-Verlag New York, Inc., 2002. 944367.
- [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, June 1999.
- [Fen94] N. Fenton. Software measurement: A necessary scientific bases,. *IEEE Transactions on Software Engineering*, 20:199–206, 1994.
- [Fle71] J L Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.
- [FN99] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

- [FN01] Fabrizio Fioravanti and Paolo Nesi. Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Trans. Software Eng.*, 27(12):1062–1084, 2001.
- [FP97] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.
- [FSJ99] Mohamed Fayad, Goug Schmidt, and Ralph Johnson. *Building Applications Frameworks: Object-oriented Foundations of FrameworkDesign*. Wiley Computer Publishing, 1999.
- [FTC07] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of feature envy bad smells. In *International Conference on Software Maintenance*, pages 519–520. IEEE, Oct. 2007.
- [FTSC11] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 1037–1039, New York, NY, USA, 2011. ACM.
- [FXC06] FXCop. [http://msdn.microsoft.com/en-us/library/bb429476\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx), June 2006. [Accessed: 2010-04-06].
- [GAA01] Yann-Gaël Gueheneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. *39th International Conference on Technology of Object-Oriented Languages and Systems*, pages 296–305, 2001.
- [GDMR04] Tudor Girba, Stéphane Ducasse, Radu Marinescu, and Daniel Ratiu. Identifying entities that change together. In *9th IEEE Workshop on Empirical Studies of Software Maintenance*, Chicago, 2004.
- [GFGP06] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of code clones and change couplings. In *9th International Conference of Fundamental Approaches to Software Engineering*, number 3922 in Lecture Notes in Computer Science, pages 411–425. Springer, 2006.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.
- [Gon08] Iker Gondra. Applying machine learning to software fault-proneness prediction. *J. Syst. Softw.*, 81(2):186–195, 2008.
- [GOPR01] Marcela Genero, José A. Olivás, Mario Piattini, and Francisco P. Romero. Using metrics to predict oo information systems maintainability. In *CAiSE*, pages 388–401, 2001.
- [Gro] LOOSE Research Group. iPlasma. <http://loose.upt.ro/iplasma>. [Accessed: 2010-04-06].

- [Gué03] Yann-Gaël Guéhéneuc. *A framework for design motif traceability*. PhD thesis, École des Mines de Nantes; University of Nantes, July 2003.
- [GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin/Heidelberg, 1999.
- [Ham07] Hammurapi. <http://www.hammurapi.biz>, October 2007. [Accessed: 2010-04-06].
- [HCN98] Rachel Harrison, Steve Counsell, and Reuben V. Nithi. Coupling metrics for object-oriented design. In *IEEE METRICS*, pages 150–157, 1998.
- [IEE05] Computer Society IEEE. *Guide to the Software Engineering Body of Knowledge: 2004 Edition - SWEBOOK*. 2005.
- [Int08] Intooitus. Incode infusion. <http://www.intooitus.com/>, 2008. [Accessed: 2010-04-06].
- [ISO01] ISO and IEC. ISO/IEC 9126-1:2001, software engineering – product quality, part 1: Quality model, 2001.
- [JAM77] G. F. Walters J. A. McCall, P. K. Richards. Factors in software quality. *Nat'l Tech.Information Service*, 1,2,3, 1977.
- [JBR00] Ivar Jacobson, Grady Booch, and James Rumbaugh. *El Proceso Unificado de Desarrollo del Software*. Addison Wesley, 2000.
- [JDe11] JDeodorant, 2011.
- [JM01] N. Juristo and A. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, 2001.
- [KAKB⁺08] Barbara Kitchenham, Hiyam Al-Khilidar, Muhammed Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering*, 13:97–121, February 2008.
- [Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Signature Series. Addison-Wesley Professional, August 2004.
- [KP96] B. Kitchenham and S. L. Pfleeger. Software quality: the elusive target. *IEEE Software*, 1:12–21, 1996.
- [KPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *16th Working Conference on Reverse Engineering (WCRE)*, pages 75–84. IEEE Computer Society, 2009.
- [KRW07] Douglas Kirk, Marc Roper, and Murray Wood. A heuristic-based approach to code-smell detection. In *1st Workshop on Refactoring Tools [DC07]*, pages 55–56.

- [KSW99] Colin Kirsopp, Martin J. Shepperd, and Steve Webster. An empirical study into the use of measurement to support oo design evaluation. In *IEEE METRICS*, pages 230–241, 1999.
- [KVG09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari A. Sahraoui. A bayesian approach for the detection of code and design smells. In *International Conference on Quality Software*, pages 305–314. IEEE Computer Society, 2009.
- [LA97] D. Lamb and J. Abounader. Data model for object-oriented design metrics. Technical report, Department of Computing and Information Science. Queens’s University., 1997.
- [LC03] Carlos López and Yania Crespo. Definición de un soporte estructural para abordar el problema de la independencia del lenguaje en la definición de refactorizaciones. Technical Report DI-2003-03, Departamento de Informática. Universidad de Valladolid, sep 2003.
- [LCMM09] Carlos López, Yania Crespo, Esperanza Manso, and Raúl Marticorena. Evaluación de código mediante múltiples intervalos de métricas. *Revista de Procesos y Métricas*, 6(1):19–30, July 2009.
- [LCMM11] Carlos López, Yania Crespo, Raúl Marticorena, and Esperanza Manso. Plugin de eclipse: Proceso dinámico de gestión de defectos de diseño mediante métricas. In *XVI Jornadas de Ingeniería del Software y Bases de Datos. La Coruña. ISBN: 978-84-9749-486-1*, pages 499 – 503. Universidad de la Coruña, Septiembre 2011.
- [LK77] J R Landis and G G Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.
- [LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
- [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.
- [LMC03] Carlos López, Raúl Marticorena, and Yania Crespo. Hacia una solución basada en frameworks para la definición de refactorizaciones con independencia del lenguaje. In Ernesto Pimentel, Nieves R. Brisaboa, and Jaime Gómez, editors, *VIII Jornadas Ingeniería del Software y Bases de Datos (JISBD 2003), Alicante, Spain. ISBN 84-688-3836-5*, pages 251–262, nov 2003.
- [LMC04] Carlos López, Raúl Marticorena, and Yania Crespo. Model language and framework support for refactoring reuse. In *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004, Madrid, Spain, ISBN 3-540-22335-5*, page Poster, jul 2004.

- [LMC06] Carlos López, Raúl Marticorena, and Yania Crespo. Caracterización de refactorizaciones para la implementación en herramientas. In *XI Jornadas de Ingeniería del Software y Bases de Datos, Sitges, Barcelona, 2006*. ISBN:84-95999-99-4, pages 538–543, oct 2006.
- [LMC10] Carlos López, Esperanza Manso, and Yania Crespo. The identification of anomalous code measures with conditioned interval metrics. In *13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2010) Málaga, Spain, July 2010*.
- [LMC11] Carlos López, Esperanza Manso, and Yania Crespo. Un caso de estudio sobre la identificación de valores umbrales para medidas de código. In *XVI Jornadas de Ingeniería del Software y Bases de Datos.. La Coruña*. ISBN: 978-84-9749-486-1., pages 471–485. Universidad de la Coruña, Septiembre 2011.
- [LMC12] Carlos López, Esperanza Manso, and Yania Crespo. Evaluación de la eficiencia de métodos de identificación del defecto de diseño godclass. In *XVII Jornadas de Ingeniería del Software y Bases de Datos.Almería*. ISBN: 978-84-1587-28-9. Universidad de Almería, Septiembre 2012.
- [LMCP06] Carlos López, Raúl Marticorena, Yania Crespo, and Javier Pérez. Towards a language independent refactoring framework. In *1st ICSSOFT 06 International Conference on Software and Data Technologies. Setubal, Portugal*. ISBN: 972-8865-69-4, volume 1, pages 165–170, sep 2006.
- [Lóp] Carlos López. Subversion, git & mercurial hosting, project tracking, trac hosting & agile tools - /src - girovisor - girovisor.
- [Lóp12a] Carlos López. Datos de experimentación en ingeniería del software relacionados con la detección de defectos de diseño mediante métricas de código, 2012. <http://giro.infor.uva.es/Publications/2012/Lop12>.
- [Lóp12b] Carlos López. Prototipos de herramientas software relacionados con la detección de defectos de diseño mediante métricas de código, 2012. <http://giro.infor.uva.es/Publications/2012/Lop12a>.
- [LS07] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, July 2007.
- [LWN07a] Ángela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Assessing the impact of bad smells using historical information. In *9th 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 *4rd International Workshop on Mining Software Repositories*, May 2007.

- [Man09] Esperanza Manso. *Estudio empírico para la validación de indicadores de la reusabilidad de diagramas de clases UML*. PhD thesis, Universidad de Valladolid, 2009.
- [Mar94] Robert Martin. OO design quality metrics; an analysis of dependencies, October 1994.
- [Mar02a] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, October 2002.
- [Mar02b] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, October 2002.
- [Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
- [McC76] Tomas McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
- [MCL05] Raúl Marticorena, Yania Crespo, and Carlos López. Soporte de métricas con independencia del lenguaje para la inferencia de refactorizaciones. In *X Jornadas Ingeniería del Software y Bases de Datos (JISBD 2005)*, Granada, Spain ISBN: 84-9732-434-X, pages 59–66, sep 2005.
- [Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, May 2007.
- [MG10] Radu Marinescu and George Ganea. incode.rules: An agile approach for defining and checking architectural constraints. In *Proceedings of the Proceedings of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing, ICCP '10*, pages 305–312, Washington, DC, USA, 2010. IEEE Computer Society.
- [MGDL10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, January/February 2010.
- [MGF07] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007. Journal.
- [MGL⁺09] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, May 2009.
- [MGV10] Radu Marinescu, George Ganea, and Ioana Verebi. Incode: Continuous quality assessment and improvement. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering, CSMR '10*, pages 274–275, Washington, DC, USA, 2010. IEEE Computer Society.

- [MHVG08] Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. Refactorings of design defects using relational concept analysis. In Raoul Medina and Sergei Obiedkov, editors, *4th International Conference on Formal Concept Analysis*, pages 289–304. Springer-Verlag, February 2008.
- [MLC03] Raúl Marticorena, Carlos López, and Yania Crespo. Refactorizaciones de especialización en cuanto a genericidad. definición para una familia de lenguajes y soporte basado en frameworks. In *III Jornadas de Programación y Lenguajes (PROLE 2003)*, Alicante, Spain., pages 75–89, nov 2003.
- [MLC05a] Raúl Marticorena, Carlos López, and Yania Crespo. Parallel inheritance hierarchy: Detection from a static view of the system. In *6th International Workshop on Object Oriented Reengineering (WOOR)*, Glasgow, UK., page 6, jul 2005. <http://smallwiki.unibe.ch/woor/workshopparticipants/>.
- [MLC05b] Raúl Marticorena, Carlos López, and Yania Crespo. Parallel inheritance hierarchy: Detection from a static view of the system. In *6th International Workshop on Object Oriented Reengineering (WOOR)*, Glasgow, UK., page 6, July 2005. <http://smallwiki.unibe.ch/woor/workshopparticipants/>.
- [MLC06a] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *7th ECOOP International Workshop on Object-Oriented Reengineering (WOOR)*. Nantes, France. <http://smallwiki.unibe.ch/woor/>, page 6, jul 2006.
- [MLC06b] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *7th International Workshop on Object-Oriented Reengineering*, page 6, Nantes, France, July 2006.
- [MLC07] Raúl Marticorena, Carlos López, and Yania Crespo. Definición de un proceso para la contrucción de refactorizaciones. In *XII Jornadas Ingeniería del Software y Bases de Datos (JISBD 2007)*, Zaragoza, Spain, pages 361–367, sep 2007.
- [Moo05] Daniel L. Moody. Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data & Knowledge Engineering*, 55(3):243–276, December 2005.
- [Mor01] S. Morasca. *Handbook of Software Engineering and Knowledge Engineering. Chapter 2: Software Measurement*. World Scientific, 2001.
- [MSPC11] Alberto Mendo, Javier Sobrino, Javier Pérez, and Yania Crespo. Evaluación de herramientas de detección y corrección de defectos de diseño. Technical Report 2011/02, Grupo GIRO, Departamento de Informática, Universidad de Valladolid, March 2011.
- [Mun05] Matthew James Munro. Product metrics for automatic identification of “bad smell” design problems in java source-code. *Software Metrics, 2005. 11th IEEE International Symposium*, pages 9 pp.–, Sept. 2005.

- [MVDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July/August 2005.
- [NB07] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software*, volume 4581/2007 of *Lecture Notes in Computer Science*, pages 228–243. Springer, Heidelberg, June 2007.
- [OCBZ09] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *3rd International Symposium on Empirical Software Engineering and Measurement, ESEM'09*, pages 390–400, Washington, DC, USA, 2009. IEEE Computer Society.
- [OKAG10] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Numerical signatures of antipatterns: An approach based on b-splines. In *14th European Conference on Software Maintenance and Re-engineering*. IEEE Computer Society, March 2010.
- [OMG06] OMG. Software process engineering meta-model 2.0, 2006.
- [OS09] Open-Source. R, 1997- 2009.
- [PG02] M.G. Piattini and F. O. García. *Calidad en el desarrollo y mantenimiento del software*. Ra-Ma, 2002.
- [Pig96] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [PJ88] Meilir Page-Jones. *The practical guide to structured systems design: 2nd edition*. Yourdon Press, Upper Saddle River, NJ, USA, 1988.
- [PLMM11] Javier Pérez, Carlos López, Naouel Moha, and Tom Mens. A classification framework and survey for design smell management. Technical Report 2011/01, Grupo GIRO, Departamento de Informática, Universidad de Valladolid, March 2011.
- [PMD09] PMD. <http://pmd.sourceforge.net>, June 2009. [Accessed: 2010-04-06].
- [Pér11] Javier Pérez. *Refactoring Planning for Design Smell Correction in Object-Oriented Software*. PhD thesis, ETSII, University of Valladolid, July 2011.
- [Pre05] Roger S. Pressman. *Ingeniería del software : un enfoque práctico*. McGraw-Hill, 6^a edition, 2005.
- [PW90] Adam A. Porter and Richard W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Softw.*, 7(2):46–54, 1990.

- [Qui86] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.
- [Qui92] J. Ross Quinlan. *C4.5: Programs for Machine Learning (Morgan Kaufmann Series in Machine Learning)*. Morgan Kaufmann, 1 edition, October 1992.
- [Rev] RevJava. <http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm>. [Accessed: 2010-04-06].
- [RH09] Per Runeson and Martin Host. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, 2009.
- [Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, April 1996.
- [Rob02] C. Robson. *Real World Research - A Resource for Social Scientists and Practitioner-Researchers*. Blackwell Publishing, Malden, second edition, 2002.
- [Roo] Roodi. <http://roodi.rubyforge.org>. [Accessed: 2010-04-06].
- [RR08] A. Ananda Rao and K. Narendar Reddy. Detecting bad smells in object oriented design using design change propagation probability matrix. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, March 2008.
- [RSG08] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *International workshop on Mining Software Repositories*, pages 35–38, New York, NY, USA, 2008. ACM.
- [RSS⁺04] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: Smart analysis based error reduction. *ACM SIGSOFT Software Engineering Notes*, 29(4):243–251, 2004.
- [Rut] Kevin Rutherford. Reek. <http://wiki.github.com/kevinrutherford/reek>. [Accessed: 2010-04-06].
- [SB05] Patrick E ShROUT and Melissa D Begg. Fleiss, Joseph I. In *Encyclopedia of Biostatistics*, pages–. John Wiley & Sons, Ltd, 2005.
- [SC88] S. Siegel and N.J. Castellan. *Non-parametric statistics for the behavioural sciences (2nd ed)*. Mc Graw Hill, 1988.
- [Sem07] SemmleCode. <http://semml.com/semmlcode>, October 2007. [Accessed: 2010-04-06].
- [SGM00] Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its

- automation? In *International Conference on Software Maintenance*, pages 154–162, Washington, DC, USA, 2000. IEEE Computer Society.
- [She00] Thomas B. Sheridan. Function allocation: algorithm, alchemy or apostasy? *Int. J. Hum.-Comput. Stud.*, 52(2):203–216, 2000.
- [SL08] Raed Shatnawi and Wei Li. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of Systems and Software*, 81(11):1868–1882, 2008.
- [Sli05] Stefan Slinger. Code smell detection in eclipse. Master’s thesis, Faculty of Electrical Engineering, Mathematics and Computer Science - Delft University of Technology, 2005.
- [SLMM99] Houari A. Sahraoui, Hakim Lounis, Walcélio L. Melo, and Hafedh Mili. A concept formation based approach to object identification in procedural code. *Automated Software Engineering*, 6(4):387–410, 1999.
- [SLT06] Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. A metric-based heuristic framework to detect object-oriented design flaws. In *14th IEEE International Conference on Program Comprehension*, 2006.
- [Som05] Ian Sommerville. *Ingeniería del software*. Pearson Educación, 7^a edition, 2005.
- [Sou] SourceForge. Sourceforge.net repository - [eclemma] index of /eclemma/-trunk.
- [SPD⁺05] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering*, 10(1):81–104, 2005.
- [SR06] Martin Lippert Stefan Rook. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, 2006.
- [SS07] G. Snelting and M. Streckenbach. Kaba: Automated refactoring for improved cohesion. In *1st Workshop on Refactoring Tools [DC07]*, pages 1–2.
- [SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *5th European Conference on Software Maintenance and Reengineering*, pages 30–38, Washington, DC, USA, 2001. IEEE Computer Society.
- [ST00] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions Programming Languages Systems*, 22(3):540–582, 2000.
- [STA] STAN. <http://stan4j.com>. [Accessed: 2010-04-06].

- [Sty] StyleCop. <http://code.msdn.microsoft.com/sourceanalysis>. [Accessed: 2010-04-06].
- [TCC08] Nikolaos Tsantalis, Tsantalis Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *12th European Conference on Software Maintenance and Reengineering*, pages 329–331, April 2008.
- [Tea09] Ptidej Team. DECOR. <http://www.ptidej.net/research/decor>, 2009. [Accessed: 2010-04-06].
- [Tes] Jean Tessier. Dependency finder metric tool.
- [Tes08] Jean Tessier. Dependency Finder. <http://depfind.sourceforge.net>, 2008. [Accessed: 2010-04-06].
- [Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.
- [TK04] Ladan Tahvildari and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 16(4-5):331–361, 2004.
- [TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *7th European Conference on Software Maintenance and Reengineering*, pages 91–100, Washington, DC, USA, 2003. IEEE Computer Society.
- [Tri08] Adrian Trifu. *Towards Automated Restructuring of Object Oriented Systems*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.
- [TSFB99] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and applications*, pages 47–56, New York, NY, USA, 1999. ACM.
- [TSG04] Adrian Trifu, Olaf Seng, and Thomas Genssler. Automated design flaw correction in object-oriented systems. In *8th European Conference on Software Maintenance and Reengineering*, pages 174–183. IEEE Computer Society Press, March 2004.
- [VDDR07] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.
- [VKMG09] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *Proceedings of the 16th Working Conference on Reverse Enginee-*

- ring (WCRE'09)*, pages 145–154, Los Alamitos, CA, USA, October 2009. IEEE Computer Society.
- [vM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *9th Working Conference on Reverse Engineering*, pages 97–107. IEEE Computer Society, October 2002.
- [vMvK01] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *2nd International Conference on Extreme Programming*, 2001.
- [Wai07] University of Waikato. Weka, 1999 - 2007.
- [Wak03] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [WBM91] B.J. Winer, D.R. Brown, and K.M. Michels. *Statistical principles in experimental design (3^a ed)*. Mc Graw Hill, 1991.
- [Web95] Bruce F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, February 1995.
- [WFH11] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, January 2011.
- [WP05] Bartosz Walter and Blazej Pietrzak. Multi-criteria detection of bad smells in code with UTA method. In *6th International Conference on Extreme Programming and Agile Processes in Software Engineering*, volume 3556 of *Lecture Notes in Computer Science*, pages 154–161. Springer, June 2005.
- [WRH⁺00] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*, volume 6 of *International Series in Software Engineering*. Springer, 2000.
- [XS04] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *16th International Conference on Software Engineering & Knowledge Engineering*, pages 123–128, 2004.