Universidad de Valladolid

Escuela Técnica Superior
de Ingeniería Informática

Departamento de Informática

Tesis Doctoral:

Refactorización sobre Programación Genérica en Lenguajes Orientados a Objetos

Presentada por Raúl Marticorena Sánchez para optar
al grado de doctor por la Universidad de Valladolid

Dirigida por: Dra. Yania Crespo González-Carvajal

marzo, 2013

A mi familia.

"Allí donde hay una esperanza, siempre hay una prueba"
1Q84. Libro 3.
Haruki Murakami

Agradecimientos

A mi directora de tesis, Yania Crespo, por el enorme esfuerzo realizado durante el desarrollo de la presente tesis doctoral. Este trabajo es también el fruto de muchas horas de revisión y corrección por su parte.

A mi compañero en el Grupo de Investigación GIRO y en la Universidad de Burgos, Carlos López. Han sido muchas horas de viaje a Valladolid, congresos y reuniones compartidas. Sin su ayuda y apoyo tampoco hubiese llegado hasta aquí.

También quería tener un recuerdo especial para Javier Pérez, Esperanza Manso, Bruno González, Miguel Ángel Laguna, Carmen Hernández y a los miembros del Grupo de Investigación GIRO.

A lo largo de estos años también he contado con la ayuda de mis compañeros del área de Lenguajes y Sistemas Informáticos en la Universidad de Burgos. Siempre han estado ahí para aclarar mis dudas. Gracias.

Por último, quiero agradecer el apoyo mostrado por mi familia: padres, hermanos, sobrinos y familia política. Gracias a ellos he podido llegar a la meta.

Raúl Marticorena Sánchez
Burgos, 29 de marzo de 2013

Convenciones Tipográficas

Las siguientes convenciones tipográficas son utilizadas a lo largo del presente documento:

- Nombres de lenguajes de programación, herramientas, productos, etc. están escritas en versalitas (*"small caps"*), *e.g.* Java, C#, Eclipse, Netbeans, etc.
- Nombre de refactorizaciones y patrones de diseño en versalitas y negrita, *e.g.* **Parametrizar**, **Template Method**.
- Nombres no traducidos del inglés por ser ampliamente utilizados en la literatura, en cursiva, *e.g. framework plugin*, etc. También aplicada en locuciones latinas *e.g. de facto*, *ad hoc*, etc.
- Listados y porciones de código están escritas en fuente `courier`, con las palabras clave del lenguaje en negrita *e.g.* `int x = a * b;`.
- Elementos de los metamodelos están escritos en `courier`, *e.g.* `ClassDef`, `NameSpace`, etc.
- Citas de otros autores están escritas en cursiva, entre dobles comillas, *e.g. "esto es una cita"*, y situadas en un párrafo a parte cuando son relevantes o de larga extensión.

Resumen

En el proceso de desarrollo del software, éste evoluciona y cambia de manera continua. Cuando la transición entre *desarrollo* y *evolución* no es continua y suave se denomina *"mantenimiento del software"*. Uno de los objetivos en dicho mantenimiento es reducir la progresiva degradación del software denominada entropía del software.

Para reducir dicha entropía, una de las labores fundamentales en el mantenimiento perfectivo es la reestructuración del código fuente. El término utilizado cuando nos referimos a reestructuración en lenguajes orientados a objetos, se denomina refactorización, como traducción del término inglés *Refactoring*. Dentro de las líneas abiertas de investigación en refactorización, cobra importancia el lograr una cierta independencia del lenguaje de programación, tanto en la definición de refactorizaciones como en su posterior implementación.

Por otro lado, hay ciertas características de los lenguajes de programación que no han sido abordadas en los trabajos realizados en refactorización. En concreto, los catálogos de refactorizaciones más utilizados en la actualidad, no abordan en particular la programación genérica y el uso de clases genéricas.

La presente Tesis Doctoral aborda el problema de la definición e implementación de refactorizaciones sobre programación genérica en lenguajes orientados a objetos. Se trabaja con el objetivo de lograr una cierta independencia en la definición de refactorizaciones, y sobre refactorizaciones centradas en las clases genéricas y sus propiedades particulares.

Como resultado, se propone un metamodelo que permite definir los conceptos básicos de los lenguajes de programación orientados a objetos, con especial hincapié sobre las características vinculadas a la programación genérica. A partir de la información disponible en este metamodelo, se propone una plantilla para la definición de refactorizaciones y el uso de un lenguaje de especificación. Se busca obtener el mayor grado de independencia del lenguaje en dichas definiciones, pero por otro lado, eliminar cierta subjetividad de la que adolecen otras definiciones. Con esta tesis, se describe un nuevo catálogo de refactorizaciones, centradas sobre programación genérica, validando la adecuación de la solución planteada.

Para dar soporte a la implementación de la construcción y ejecución de las refactorizaciones se propone una arquitectura basada en *frameworks*, a partir de la cual se construye un prototipo. Esto permite un doble enfoque: general para los lenguajes de la familia de lenguajes de programación orientados a objetos, y particular para el lenguaje objetivo para el que se implementan los catálogos de refactorizaciones. Finalmente se valida la solución propuesta sobre un lenguaje concreto como Java.

En resumen, se aporta un nuevo enfoque a la hora de abordar las refactorizaciones, que mejora algunas de las deficiencias encontradas en la actualidad.

Palabras clave: mantenimiento del software, refactorización, independencia del lenguaje, programación genérica, genericidad.

Índice general

Índice de figuras VII

Índice de tablas IX

Índice de códigos XI

1 Introducción 1
- 1.1 Mantenimiento en la ingeniería del software . . . 1
- 1.2 Evolución del software: de la reestructuración a la refactorización . . . 3
- 1.3 Refactorización en la evolución del software . . . 6
- 1.4 Dependencia del lenguaje de programación en refactorización . . . 8
- 1.5 Hacia una cierta independencia del lenguaje . . . 8
- 1.6 Genericidad y refactorización: ¿cuál es la situación actual? . . . 9
- 1.7 Objetivos del trabajo . . . 9
- 1.8 Resultados esperados . . . 10
- 1.9 Estructura del documento . . . 11

2 Estado del arte 13
- 2.1 Refactorización . . . 13
  - 2.1.1 Transformación de programas . . . 13
  - 2.1.2 Taxonomía de transformaciones . . . 14
  - 2.1.3 Refactorización: definición y características . . . 17
  - 2.1.4 Catálogos de refactorizaciones . . . 21
  - 2.1.5 Refactorización en aplicaciones y entornos de desarrollo integrados . . . 25
  - 2.1.6 Caracterización . . . 28
  - 2.1.7 Proceso . . . 30
  - 2.1.8 Independencia del lenguaje en refactorización . . . 31
- 2.2 Genericidad en LPOO estáticamente tipados . . . 35
  - 2.2.1 Tipos paramétricos y acotación de tipo . . . 35
  - 2.2.2 Clases y programación genérica . . . 37
  - 2.2.3 Integración de la genericidad en lenguajes de programación orientados a objetos . . . 37
  - 2.2.4 Relevancia de la programación genérica: refactorización en genericidad 48

3 Independencia del lenguaje en refactorización **51**
3.1 MOON como marco de referencia . . . 53
3.1.1 De la gramática MOON al metamodelo MOON . . . 55
3.1.2 Tipos y clases . . . 56
3.1.3 Entidades . . . 57
3.1.4 Métodos . . . 59
3.1.5 Genericidad . . . 60
3.1.6 Herencia . . . 62
3.1.7 Expresiones e instrucciones . . . 63
3.1.8 Comentarios de código . . . 68
3.1.9 Jerarquía principal de MOON . . . 68
3.1.10 Conclusiones al metamodelo MOON . . . 70
3.2 Arquitectura general . . . 72

4 Definición, caracterización y proceso **75**
4.1 Estructura en la definición de refactorizaciones . . . 75
4.1.1 Estado del arte en la estructura de la definición de refactorizaciones . 76
4.1.2 Una propuesta para la estructura de la definición de refactorizaciones . 81
4.2 Caracterización de refactorizaciones . . . 85
4.2.1 Caracterización propuesta . . . 86
4.3 Proceso de definición de refactorizaciones . . . 92
4.3.1 Situación previa al estudio . . . 93
4.3.2 Elementos del proceso: roles, tareas y productos . . . 93
4.3.3 Proceso incremental propuesto para la definición de refactorizaciones . 96

5 Catálogo de refactorizaciones sobre genericidad **101**
5.1 Descripción de refactorizaciones con Alf . . . 102
5.2 Funciones básicas . . . 105
5.2.1 **DependantDownFormalPar** . . . 105
5.2.2 **DependantUpFormalPar** . . . 106
5.2.3 **RealSubstitutionsOfFormalPar** . . . 108
5.2.4 **CollectSignaturesToAdd** . . . 108
5.2.5 **CollectMethodIntersection** . . . 108
5.2.6 **CollectEntitiesWithType** . . . 108
5.2.7 **CollectMethodsUsedByEntity** . . . 109
5.2.8 **CollectDownRealSubstitutions** . . . 109
5.2.9 **CollectPropertiesUsedByEntity** . . . 109
5.2.10 **DependantDescendantsProvidersBoundW** . . . 109
5.2.11 **ParameterizeUniverse** . . . 109
5.2.12 **EquivalentEntities** . . . 110
5.2.13 **GDD** . . . 110
5.2.14 **GDI** . . . 110
5.2.15 **GenericDependant** . . . 110
5.2.16 **ExpressionsAssignedToEntity** . . . 110

5.2.17 **CallExprUsedByEntity** . . . 110
5.2.18 **CallExprUsingEntity** . . . 110
5.3 Catálogo en generalización . . . 111
5.3.1 Refactorización Parametrizar *(Parameterize)* . . . 111
5.3.2 Refactorización Generalizar acotación *(Generalize bound)* . . . 118
5.3.3 Refactorización Eliminar signaturas en cláusula ***tal que*** *(Remove signatures in where clause)* . . . 124
5.4 Catálogo en especialización . . . 129
5.4.1 Refactorización Reemplazar parámetro formal por tipo completo (*Replace formal parameter with complete type*) . . . 129
5.4.2 Refactorización Especializar acotación (*Specialize bound*) . . . 136
5.4.3 Refactorización Añadir signaturas en cláusulas ***tal que*** *(Add signatures in where clause)* . . . 140
5.5 Caracterización del catálogo de refactorizaciones en genericidad . . . 145

6 Solución basada en frameworks **147**
6.1 *Frameworks* como solución . . . 148
6.1.1 Concepto de *framework* . . . 148
6.1.2 Ventajas y desventajas del uso de *frameworks* . . . 149
6.1.3 Clasificación de *frameworks* . . . 150
6.1.4 Documentación de *frameworks* . . . 151
6.1.5 *Frameworks* en refactorización . . . 151
6.2 Representación del código . . . 152
6.3 Construcción y ejecución de refactorizaciones . . . 155
6.3.1 Núcleo del motor de refactorizaciones . . . 155
6.3.2 Construcción y ejecución estática . . . 158
6.3.3 Evolución hacia *framework* de caja negra . . . 159
6.3.4 Construcción y ejecución dinámica basada en XML . . . 160
6.3.5 Elementos XML de una refactorización . . . 161
6.3.6 Asistentes en la construcción de refactorizaciones y uso de reflexión . . . 162
6.4 Regeneración del código refactorizado . . . 163
6.4.1 Regeneración a partir de representaciones intermedias . . . 164
6.4.2 Módulo de regeneración de código . . . 165
6.5 Conclusiones al uso de *frameworks* . . . 167

7 Caso de estudio **169**
7.1 Extensión de Java para el metamodelo MOON: JavaMOON . . . 170
7.1.1 Tipos y Clases . . . 170
7.1.2 Entidades . . . 171
7.1.3 Métodos . . . 174
7.1.4 Genericidad . . . 175
7.1.5 Herencia . . . 175
7.1.6 Expresiones e instrucciones . . . 176
7.1.7 Comentarios de código . . . 181

7.1.8 Anotaciones . . . 182
7.1.9 Conclusiones al metamodelo JavaMOON . . . 183
7.2 Desarrollo actual . . . 183
7.2.1 Construcción de refactorizaciones . . . 184
7.2.2 Aplicación de refactorizaciones . . . 190
7.3 Refactorizaciones incluidas . . . 191
7.3.1 Refactorizaciones básicas . . . 191
7.3.2 Refactorizaciones en genericidad . . . 192
7.3.3 Refactorizaciones en la migración de bibliotecas y *frameworks* . . . 192
7.3.4 Refactorizaciones en la evolución del lenguaje . . . 193
7.4 Regeneración del código fuente . . . 193
7.5 Análisis o matriz DAFO del prototipo. . . . 195

8 Conclusiones y líneas de trabajo futuro 197
8.1 Conclusiones generales . . . 197
8.2 Conclusiones desglosadas por objetivos . . . 198
8.3 Limitaciones y debilidades . . . 200
8.4 Líneas de trabajo futuro . . . 201
8.4.1 Aplicación de refactorizaciones definidas con MOON a otros lenguajes 201
8.4.2 Refactorización en la migración del software . . . 202
8.4.3 Refactorización del software y evolución de los lenguajes de programación 203

Apéndices 206

A Sintaxis concreta de MOON 209

B Caracterización del catálogo de Fowler 213
B.1 Orden en la implementación . . . 213

C Especificación de funciones en Alf 227
C.1 **DependantDownFormalPar** . . . 227
C.2 **RealSubstitutionsOfFormalPar** . . . 229
C.3 **CollectSignaturesToAdd** . . . 229
C.4 **CollectMethodIntersection** . . . 230
C.5 **DependantDescendantsProvidersBoundW** . . . 231
C.6 **CollectEntitiesWithType** . . . 233
C.7 **CollectMethodsUsedByEntity** . . . 233
C.8 **DependantUpFormalPar** . . . 234
C.9 **CollectDownRealSubstitutions** . . . 235
C.10 **CollectPropertiesUsedByEntity** . . . 236

D Especificación de Parametrizar en Alf 237
D.1 **ParameterizeUniverse** . . . 237
D.2 **EquivalentEntities** . . . 238
D.3 **GenericDependant** . . . 240

D.4 **GDD** . . . 256
D.5 **GDI** . . . 256
D.6 **ExpressionsAssignedToEntity** . . . 256
D.7 **CallExprUsedByEntity** . . . 258
D.8 **CallExprUsingEntity** . . . 259
D.9 **AddFormalParameterInCandidateClassesAction** . . . 261
D.10 **ChangeGDDEntitiesAction** . . . 262
D.11 **ChangeGDIEntitiesAction** . . . 263
D.12 **ChangeClientsOfGDIEntitiesAction** . . . 266

E Publicaciones **267**
E.1 Congresos y talleres . . . 267
E.2 Publicaciones y relación con los capítulos . . . 269

Bibliografía **273**

Índice de figuras

2.1 Modelo de caracterización de refactorizaciones 29
2.2 Arquitectura de MOOSE [Ducasse et al., 2001] 32

3.1 Tipos y clases 57
3.2 Entidades 58
3.3 Métodos 59
3.4 Genericidad 60
3.5 Herencia 63
3.6 Expresiones 64
3.7 Constantes 65
3.8 Instrucciones 66
3.9 Soluciones en las invocaciones con expresiones 67
3.10 Jerarquía principal 69
3.11 Arquitectura general 73

4.1 Refactorización **EXTRACT CLASS** 79
4.2 Roles en el proceso de definición y construcción de refactorizaciones 93
4.3 Productos en el proceso de definición y construcción de refactorizaciones . . . 95
4.4 Proceso propuesto para la definición, construcción y aplicación de refactorizaciones 96
4.5 Definición incremental de refactorizaciones en el análisis 97

5.1 Flujo de trabajo usando ALF en la definición de refactorizaciones 104
5.2 Parametrizar: Reemplazar tipo completo por un parámetro formal 112
5.3 Generalizar acotación 119
5.4 Eliminar signaturas en cláusula *tal que* 124
5.5 Reemplazar parámetro formal por tipo completo 130
5.6 Especializar acotación (*Specialize bound*) 136
5.7 Añadir signaturas en cláusula *tal que* 140

6.1 Fase de análisis de código y generación del modelo 154
6.2 Motor de refactorizaciones 156
6.3 Ejecución dinámica de refactorizaciones 160
6.4 Regeneración del código 165
6.5 Diagrama de clases de regeneración del código 166

7.1 Tipos y clases en JavaMOON . . . 171
7.2 Tipos primitivos en JavaMOON . . . 172
7.3 Entidades en JavaMOON . . . 173
7.4 Entidades artificiales en JavaMOON . . . 173
7.5 Métodos en JavaMOON . . . 174
7.6 Genericidad en JavaMOON . . . 175
7.7 Herencia en JavaMOON . . . 176
7.8 Expresiones en JavaMOON . . . 177
7.9 Operaciones en JavaMOON . . . 178
7.10 Constantes en JavaMOON . . . 178
7.11 Jerarquía paralela de llamadas a expresiones y uso de constantes manifiestas en JavaMOON . . . 179
7.12 Instrucciones en JavaMOON . . . 180
7.13 Comentarios en Java . . . 181
7.14 Anotaciones en JavaMOON . . . 182
7.15 Paso 1: Descripción de la refactorización . . . 185
7.16 Paso 2: Entradas . . . 186
7.17 Paso 3: Precondiciones . . . 187
7.18 Paso 4: Acciones . . . 188
7.19 Paso 7: Resumen . . . 189
7.20 Visualización de refactorizaciones disponibles . . . 190
7.21 Regeneración de código Java . . . 194

B.1 Refactorizaciones relacionadas en el grafo Composing Methods (7 de 9) . . . 214
B.2 Refactorizaciones relacionadas en el grafo Moving Features Between Objects (6 de 8) . . . 215
B.3 Refactorizaciones relacionadas en el grafo Organizing Data (5 de 16) . . . 216
B.4 Refactorizaciones relacionadas en el grafo Making Method Calls Simpler (4 de 15) . . . 219
B.5 Refactorizaciones relacionadas en el grafo Dealing with Generalization (9 de 12) . . . 220

Índice de tablas

2.1 Resumen de características de transformaciones de traducción 16
2.2 Resumen de características de transformaciones de reescritura 17
2.6 Propiedades generales sobre genericidad en los LPOO principales 47

4.1 Estructura propuesta en [Marticorena et al., 2003] para la definición de refactorizaciones 81
4.2 Refactorización **Push Down Field** de acuerdo a la estructura definida en [Marticorena et al., 2003] 82

5.1 Catálogo de refactorizaciones sobre genericidad 102
5.2 Orden parcial de mayor a menor complejidad del grupo de refactorizaciones en genericidad 146

7.1 Matriz DAFO del prototipo 196

8.1 Comparativa entre rejuvenecimiento y refactorización (tomada de [Pirkelbauer et al., 2010]) 204

B.1 Orden parcial de mayor a menor complejidad del grupo Composing Methods 215
B.2 Orden parcial de mayor a menor complejidad del grupo Moving Features Between Objects 216
B.3 Orden parcial de mayor a menor complejidad del grupo Organizing Data . 217
B.4 Orden parcial de mayor a menor complejidad del grupo Simplifying Conditional Expressions 218
B.5 Orden parcial de mayor a menor complejidad del grupo Making Method Calls Simpler 219
B.6 Orden parcial de mayor a menor complejidad del grupo Dealing with Generalization 221
B.7 Orden parcial de mayor a menor complejidad del catálogo en [Fowler, 1999] . 222

E.1 Referencias cruzadas entre publicaciones y módulos 271

Índice de códigos

4.1.1 Refactorización **Push Down Field** utilizando Alf como lenguaje para su descripción . . . 83
5.3.1 Predicado `NotExistsFormalParWithEqualName` . . . 113
5.3.2 Predicado `TargetEntityWithNotVarType` . . . 113
5.3.3 Predicado `TargetEntityNotRecursive` . . . 114
5.3.4 Predicado `IsRecursivePositive` . . . 114
5.3.5 Predicado `IsRecursiveNegative` . . . 114
5.3.6 Predicado `NotExistsInBothSets` . . . 115
5.3.7 Predicado `NotAssignmentWithConstant` . . . 115
5.3.8 Acción `ParameterizeAction` . . . 116
5.3.9 Acción `RemoveCastAction` . . . 116
5.3.10 Acción `RemoveCastFromResultAction` . . . 117
5.3.11 Refactorización **Parametrizar** . . . 118
5.3.12 Ejemplo con código MOON del análisis de **Generalizar acotación** . 119
5.3.13 Predicado `IsSubtype` . . . 122
5.3.14 Predicado `IsGeneralizable` . . . 122
5.3.15 Acción `GeneralizeAction` . . . 123
5.3.16 Refactorización **Generalizar acotación** . . . 124
5.3.17 Ejemplo con código MOON del análisis de **Eliminar signaturas en cláusula tal que** . . . 125
5.3.18 Acción `RemoveMethodsInWhereClause` . . . 127
5.3.19 Predicado `ForAllEquivalentBoundWMethodsUsed` . . . 127
5.3.20 Refactorización **Eliminar signaturas en cláusula tal que** . . . 128
5.4.21 Ejemplo con código MOON del análisis de **Reemplazar parámetro formal por tipo completo** . . . 130
5.4.22 Predicado `IsComplete` . . . 132
5.4.23 Predicado `ForAllSubstitutionsSameType` . . . 132
5.4.24 Predicado `ForAllBoundsSameType` . . . 133
5.4.25 Acción `ReplaceFormalParInAncestorsAndProvidersAction` . . . 134
5.4.26 Acción `RemoveFormalParInDependantDownAction` . . . 134
5.4.27 Refactorización **Reemplazar parámetro formal por tipo completo** . . . 135
5.4.28 Ejemplo con código MOON del análisis de **Especializar acotación** . 136
5.4.29 Predicado `IsSpecializable` . . . 138

5.4.30 Acción SpecializeAction . . . 139
5.4.31 Refactorización **SpecializeBoundRefactoring** . . . 140
5.4.32 Ejemplo con código MOON del análisis de **Añadir signaturas en cláusula tal que** . . . 142
5.4.33 Añadir signaturas en cláusula *tal que*: Situación a) . . . 142
5.4.34 Añadir signaturas en cláusula *tal que*: Situación b) . . . 142
5.4.35 Acción AddMethodsInWhereClause . . . 143
5.4.36 Predicado ForAllSignatureIsIncludedInSubstitutions . . . 144
5.4.37 Refactorización **Añadir signaturas en cláusula tal que** . . . 145
6.3.1 Método plantilla en la clase Refactoring . . . 157
6.4.2 Interfaz Visitor . . . 166
7.2.1 Fichero con la definición de la refactorización Rename Class . . . 188
C.1.1 Función DependantDownFormalPar . . . 227
C.1.2 Función CollectDependantDownFormalPar . . . 228
C.2.3 Función RealSubstitutionsOfFormalPar . . . 229
C.3.4 Función CollectSignaturesToAdd . . . 229
C.4.5 Función CollectMethodIntersection . . . 230
C.5.6 Función DependantDescendantsProvidersBoundW . . . 231
C.5.7 Función CollectDependantProvidersBoundW . . . 232
C.5.8 Función CollectDependantDescendantsBoundW . . . 232
C.6.9 Función CollectEntitiesWithType . . . 233
C.7.10 Función CollectMethodsUsedByEntity . . . 233
C.8.11 Función DependantUpFormalPar . . . 234
C.8.12 Función CollectDependantUpFormalPar . . . 235
C.9.13 Función CollectDownRealSubstitutions . . . 235
C.10.14 Función CollectPropertiesUsedByEntity . . . 236
D.1.1 Función ParameterizeUniverse . . . 237
D.2.2 Función EquivalentEntities . . . 238
D.2.3 Función EquivalentProperties . . . 239
D.2.4 Función EquivalentUpProperties . . . 239
D.2.5 Función EquivalentDownProperties . . . 240
D.3.6 Función GenericDependant . . . 241
D.3.7 Función Step1 . . . 242
D.3.8 Función Step2 . . . 242
D.3.9 Función Step3 . . . 243
D.3.10 Función Step4 . . . 243
D.3.11 Función Step6 . . . 244
D.3.12 Función Step6-7substep . . . 244
D.3.13 Función Step6bis . . . 245
D.3.14 Función Step7 . . . 245
D.3.15 Función Step7CheckArguments . . . 246
D.3.16 Función Step9 . . . 246
D.3.17 Función Step9OptionA . . . 247
D.3.18 Función Step9OptionB . . . 248

D.3.19 Función Step9OptionC . . . 248
D.3.20 Función Step10 . . . 249
D.3.21 Función Step10CheckArguments . . . 250
D.3.22 Función Step10OptionA . . . 250
D.3.23 Función Step10OptionB . . . 251
D.3.24 Función Step10 . . . 251
D.3.25 Función AddEntityInProperGDX . . . 252
D.3.26 Función Gen . . . 252
D.3.27 Función GetCStar . . . 253
D.3.28 Función GetEntitiesCStar . . . 253
D.3.29 Función InGDX . . . 253
D.3.30 Función ProcessExprAsFunctionResult . . . 254
D.3.31 Función ProcessLeftExpressions . . . 254
D.3.32 Función EquivalentDownEntities . . . 255
D.4.33 Función GDD . . . 256
D.5.34 Función GDI . . . 256
D.6.35 Función ExpressionsAssignedToEntity . . . 257
D.6.36 Función VisitInstructionsInBodyMethod . . . 257
D.7.37 Función CallExprUsedByEntity . . . 258
D.7.38 Función VisitInstructionsInBodyMethod . . . 258
D.7.39 Función VisitCallExpr . . . 259
D.8.40 Función CallExprUsingEntity . . . 259
D.8.41 Función VisitInstructionsInBodyMethod . . . 260
D.8.42 Función VisitCallExpr . . . 261
D.9.43 Acción AddFormalParameterInCandidateClassesAction . . . 261
D.9.44 Acción ChangeInheritanceClausesAction . . . 262
D.9.45 Acción AddTypeInInheritanceClauseAction . . . 262
D.10.46 Acción ChangeGDDEntitiesAction . . . 262
D.11.47 Acción ChangeGDIEntitiesAction . . . 263
D.11.48 Acción ChangeClientsOfEntityAction . . . 265
D.11.49 Función CreateNewParametricType . . . 265
D.12.50 Acción ChangeClientsOfGDIEntitiesAction . . . 266

CAPÍTULO 1

INTRODUCCIÓN

En este capítulo se realizará una revisión del concepto de mantenimiento y evolución del software, y cómo la refactorización es un elemento fundamental en estos procesos. Se aborda el problema de la independencia del lenguaje en refactorización y se introduce la genericidad como una característica no suficientemente estudiada en el campo de las refactorizaciones, estableciendo así el conjunto de objetivos y de resultados esperados de esta tesis doctoral.

1.1. Mantenimiento en la ingeniería del software

En el proceso de desarrollo del software, partiendo de una primera entrega o versión, éste evoluciona y cambia de manera continua, adaptándose a nuevas necesidades –cambios en los requisitos funcionales y no funcionales. Este proceso habitualmente se ajusta a un modelo de ciclo de vida en espiral. Cuando la transición entre *desarrollo* y *evolución* no es continua y suave se denomina *"mantenimiento del software"* [Sommerville, 2010]. Así pues entendemos como mantenimiento a todo el proceso de cambios que sufre el software a partir de su primera entrega al cliente [Bennett and Rajlich, 2000], donde generalmente los equipos de desarrollo y mantenimiento no son los mismos.

En [Sommerville, 2010] se distinguen tres tipos de mantenimiento: reparación de errores, adaptaciones al nuevo entorno y adición de nueva funcionalidad. Otra clasificación establece que el mantenimiento puede ser correctivo, adaptativo, perfectivo y, además, preventivo [Bennett and Rajlich, 2000]. En este último, una de las definiciones aceptadas habla de tareas de mantenimiento con el fin de mejorar la estructura interna y rendimiento, conservando el comportamiento observable. Esta definición parece un claro precedente de la definición generalmente aceptada de refactorización que se recoge en la Sec. 1.3. Como se puede apreciar, el mantenimiento puede ser visto desde diferentes perspectivas, y dentro de esas lecturas de

cambio evolutivo, aparece ya el concepto de refactorización que se detallará posteriormente (Sec. 1.3). Sin embargo se debe señalar que los porcentajes de esfuerzo dedicados a las tareas relativas a los diferentes tipos de mantenimiento difieren mucho entre sí, quedando con el menor porcentaje de dedicación el mantenimiento preventivo [Bennett and Rajlich, 2000][1].

En cualquier desarrollo, una vez el software está "finalizado" se deben considerar los costes de mantenimiento que surgirán de la evolución del mismo, formando al final gran parte de los costes totales del proyecto [Roberts, 1999b]. Como se señala en [Sommerville, 2010], a menudo los costes de mantenimiento del software a medida superan los costes iniciales de desarrollo. Es muy importante considerar, desde la fase de desarrollo, un desarrollo *orientado a facilitar el mantenimiento*, puesto que esto tiene un efecto directo en la reducción de costes y en la vida práctica del producto software.

Curiosamente, muchos de los grandes errores en el software se han debido a tareas de mantenimiento como el cambio de una simple línea de código. El problema surge de no controlar los efectos de los cambios, no existiendo métodos lo suficientemente maduros que aseguren la corrección de los mismos. Este problema puede venir originado porque el mantenimiento del software no tiene la misma consideración de desafío intelectual y reto imaginativo [Brooks, 1995] que sí tiene la fase de desarrollo del mismo, considerándose una tarea rutinaria y penosa una vez construido [Sommerville, 2010].

Los problemas en mantenimiento pueden tener distintos orígenes. La ausencia de especificaciones, que aún existiendo, no forman parte del contrato de los módulos implementados, impidiendo una verificación formal. Ante cambios pequeños, el programador no repite de nuevo la ejecución de las baterías de pruebas, si existen, y tampoco sopesa cuidadosamente los posibles efectos y consecuencias de los cambios realizados.

Como resultado de todo esto, se produce en el mantenimiento del software lo que se denomina como la **entropía software**. La entropía software puede ser definida como *"la observación a lo largo del tiempo de que la estructura del software se degrada"* [Roberts, 1999b]. Dicha afirmación se puede ver apoyada en la leyes de Lehman, establecidas a partir de la observación de distintos proyectos en [Lehman, 1996]. Como resultado de las labores de mantenimiento del software, se destruye la estructura y se incrementa la entropía y el desorden del sistema. Cuanto menos tiempo se gasta en mejorar el diseño original, más y más tiempo se necesita en arreglar defectos introducidos en las primeras correcciones. Según el tiempo pasa, el sistema llega a estar peor estructurado.

En un sistema perfectamente estructurado, todo cambio está completamente localizado y acotado, y por lo tanto el coste de mantenimiento es constante. Sin embargo, en la práctica esto no ocurre, aumentando los efectos indeseados de los consiguientes arreglos, así como de los costes asociados, aumentando la entropía software.

Además, el problema de determinar para un programa qué va a cambiar en un futuro es extraordinariamente difícil. Es utópico partir de que el diseñador o programador sea clarividente y, de hecho, muchos de los defectos detectados posteriormente surgen de un cierto abuso de anticipación de necesidades futuras. La tendencia a dotar de cierta flexibilidad al

[1] Se establece un 75 % al adaptativo y perfectivo, 21 % al correctivo y sólo un 4 % al preventivo.

programa, que nunca llega a ser necesaria en la práctica, y pasar por alto ciertas propiedades del programa que finalmente se tendrán que modificar, desemboca finalmente en un sistema frágil.

En los sistemas evolutivos, el diseño es creado y mejorado mientras ciertas fuerzas se manifiestan, al igual que se expone en las metodologías ágiles. Continuamente se hace gasto de tiempo y energía para adaptar el sistema a los cambios. Para resolver estos problemas, es necesario incorporar nuevas técnicas que ayuden a minimizar dichos costes. En respuesta a estas necesidades, surgen conceptos como la reestructuración y refactorización de programas, guiando este proceso de forma más o menos automática, abaratando los costes de desarrollo y posterior mantenimiento. Este método puede llevar al éxito siempre y cuando se vea apoyado por herramientas que permitan automatizar las tareas, a un bajo coste [Roberts, 1999b].

1.2. Evolución del software: de la reestructuración a la refactorización

Podemos tomar una primera definición de evolución como *"la fase en la que se hacen cambios significativos a la arquitectura software y su funcionalidad"* [Sommerville, 2010]. La evolución del software ha sido objeto de estudio y seguimiento desde hace muchos años, como parte integral de la ingeniería del software [Sommerville, 2010].

En [Lehman and Belady, 1985] se recogen resultados de la observación de la evolución de proyectos ya realizados en los sesenta en IBM, recibiendo distintas denominaciones, inicialmente como la *dinámica de crecimiento del software*, hasta el concepto de *evolución de software* que usamos hoy en día.

Con una aproximación más actual [Lehman et al., 2000], el término se puede ver desde un doble punto de vista: el qué (causas, procesos, efectos) y el cómo (desarrollo de los mejores métodos y herramientas). Es decir, se distingue entre el problema de la evolución, y el cómo se debe gestionar dicha evolución a través de actividades de diseño, mantenimiento, reingeniería, refactorización, etc. [Cook et al., 2006].

Más aún, como propone [Sommerville, 2010], la evolución es una parte integral de la ingeniería del software que va de la mano del desarrollo, no pudiéndose considerar como una tarea independiente.

Las Leyes de Lehman, mencionadas anteriormente, identifican el conocimiento adquirido de la observación de las características comunes en la evolución de sistemas software. De las ocho leyes enunciadas en [Lehman, 1996], se extraen las siguientes[2] por estar directamente relacionadas con la parte operacional que será abordada en este trabajo:

I. Cambio continuo un programa de tipo E[3] que es usado, debe ser continuamente adaptado, o bien progresivamente será menos satisfactorio.

[2] Se mantiene la numeración romana utilizada en [Lehman, 1996].

[3] Un programa de tipo E ejecuta una actividad humana o social, propensa inherentemente al cambio [Lehman, 1980].

II. Complejidad incremental cuando un programa evoluciona, su complejidad se incrementa, salvo que se haga un trabajo para reducir dicha complejidad.

VI. Continuo crecimiento la funcionalidad de un programa debe ser continuamente incrementada para mantener al usuario satisfecho.

VII. Degradación de la calidad en los programas de tipo E se percibe que la calidad se degrada, salvo que de manera rigurosa se hagan tareas de mantenimiento y adaptación a cambios en los entornos actuales.

Como se puede concluir de estas leyes, el mantenimiento del software, en cualquiera de su vertientes, es algo intrínseco al mismo, si se quiere obtener un software de calidad. El cambio es un elemento implícito en el desarrollo del software [Sommerville, 2010], y debe afrontarse su inclusión de la forma más natural posible.

Desde el punto de vista de dicho cambio, la taxonomía propuesta en [Mens, 1999] indica que:

En diseño los cambios se realizan manualmente por un ingeniero software durante el desarrollo del proyecto. Surgen problemas como estimar los efectos del cambio, la necesidad de herramientas que asistan a la aplicación de cambios, etc.

En ejecución el software es cambiado dinámicamente mientras se ejecuta. Se necesita reflexión, evaluación parcial, computación dinámica, etc.

Desde la perspectiva de dicha clasificación, este trabajo se centrará en la evolución en diseño, contemplando sólo aquellas técnicas y herramientas que permitan modificar el producto en fases previas a su ejecución, en la fase de desarrollo. Aunque se toma como objetivo principal del trabajo el desarrollo del software, también se aplicarán en labores posteriores de mantenimiento.

En esta línea, en [Casais, 1990, Casais, 1992, Casais, 1994], se analizó la problemática de la evolución del software, desde el punto de vista de la reestructuración de programas en diseño. El concepto de reestructuración se puede definir como *"la transformación desde una forma de representación a otra al mismo nivel de abstracción, mientras se preserva el comportamiento externo del sistema (funcional y semántico)"* [Chikofsky and Cross, 1990].

Aunque inicialmente se consideró la orientación a objetos como la solución a los problemas de mantenimiento que hasta el momento se habían observado, pronto se pudo comprobar cómo aparecían nuevos problemas asociados. Los requisitos de usuario siguen siendo inestables, haciendo que los cambios se produzcan de manera continua. Las taxonomías son difíciles de establecer, y no siempre es posible situar una cierta entidad en una jerarquía de herencia. Además las clases reutilizables no surgen de la nada, sino de un proceso iterativo. Finalmente las jerarquías de herencia declaradas pueden no ser compatibles con otras jerarquías desarrolladas en paralelo. Todos estos problemas dificultan de nuevo las labores de mantenimiento, y hacen que se tenga que poner en consideración medidas correctoras en la evolución.

Como solución se identificaron cuatro categorías de soporte a la evolución en sistemas orientados a objetos [Casais, 1990, Casais, 1992]:

Sastrería adaptación de una clase cuando una relación de extensión simple no resuelve los problemas. Incluye manipulaciones como renombrados, redefinición y exportaciones selectivas, disponibles en Eiffel, o mecanismos para que una clase actúe como otra, en una jerarquía ya existente, como en Objective-C.

En general, estas operaciones se apoyan en las opciones que permite el lenguaje para hacer pequeñas adaptaciones de las clases, con los mecanismos existentes, no modificando las clases iniciales. Sin embargo pueden generar problemas: difícil integración con los sistemas de persistencia, ruptura de la estructura y ofuscación de la dependencia entre clases, alteración de la semántica y pérdida de los beneficios de la herencia.

Cirugía se produce ante cambios en el propio dominio del problema que provocan cambios traumáticos en los correspondientes módulos. Los cambios son más traumáticos puesto que se debe modificar la actual estructura, manteniendo la consistencia. El problema ya había sido estudiado en bases de datos orientadas a objetos, señalando cuatro fases:

1. Determinar las restricciones a mantener.
2. Clasificar el conjunto de primitivas de actualización a realizar.
3. Caracterizar de forma precisa sus efectos en la jerarquía y las condiciones para su aplicación.
4. Reflejar los cambios en el almacenamiento persistente.

La principal laguna de la propuesta es saber por qué y cuándo se deben aplicar las modificaciones.

Versionado seguimiento y localización de los cambios en las entidades del sistema, cuando éstas se realizan simultáneamente por diferentes personas.

Reorganización proceso de hacer grandes cambios a las bibliotecas de clases. Durante este proceso el programador intentará diferentes alternativas de diseño.

Aunque estos trabajos se centraron en la reorganización de jerarquías de herencias, algunos de sus resultados y conclusiones parciales siguen siendo vigentes hoy en día, y se desarrollaron en paralelo a la aparición del concepto de refactorización [Opdyke, 1992] en el ámbito de la programación orientada a objetos y su mantenimiento, encontrándose puntos en común con las categorías de cirugía y reorganización.

Podemos encontrar coincidencias con [Brooks, 1995] (sobre la base de los trabajos de [Mills, 1971, Baker, 1972]) donde se utiliza la metáfora del cirujano y el copiloto. Esta idea se puede ver como una anticipación a la programación por parejas defendida por XP [Beck, 1999, Beck and Andres, 2004] en la que se apunta a la importancia de razonar y discutir sobre el diseño y la solución de código aportada. Aunque el cirujano tomaba el rol de programador

principal, el copiloto podía discutir y sugerir modificaciones al código elaborado por el cirujano, obligando a una revisión y razonamiento de la solución adoptada. El trabajo de ambos, cirujano y copiloto, se veía apoyado por otros roles como el desarrollador de herramientas y el jurista del lenguaje, que daban un soporte técnico a la hora de manipular el código con las herramientas adecuadas, y con los mecanismos más precisos de los que dotaba el lenguaje de programación en uso (similar al concepto de *idioms* en la aplicación de patrones de diseño [Gamma et al., 1995]).

Aunque la propuesta se realizó hace muchos años, tiene un reflejo directo en las mejores prácticas de las metodologías ágiles y anticipaba muchos de los pilares en los que se sustenta en la actualidad la evolución del software y el soporte que se ha dado con la refactorización. Dichas metodologías ágiles encajan perfectamente con esa visión integrada del desarrollo y evolución del software como un único proceso. De hecho la visión en metodologías ágiles de un proceso de desarrollo basado en un proceso de evolución software [Sommerville, 2010], remarca la importancia y la consideración que debe darse a éste.

1.3. Refactorización en la evolución del software

Al igual que los lenguajes de programación dentro del paradigma de la programación estructurada sufrieron una evolución hacia el paradigma de la orientación a objetos, se puede ver un cierto paralelismo del concepto de reestructuración de programas, y su evolución cuando se habla de programas escritos en estos lenguajes.

El término utilizado cuando nos referimos a reestructuración en lenguajes orientados a objetos, se denomina refactorización[4], pudiéndose registrar los primeros usos del término en los trabajos de [Opdyke, 1992, Roberts, 1999b] y estableciéndose como guía definitiva el trabajo de [Fowler, 1999]. Es en este trabajo donde se recoge la definición más extendida del término:

> *"Un cambio hecho a la estructura interna del software para facilitar su comprensión y reducir el coste de mantenimiento, sin cambiar su comportamiento observable."*

Las leyes de [Lehman, 1996], anticipaban que el software tiene una tendencia a degradarse y ser incomprensible, difícil de mantener y más complejo. Esta degradación que hemos denominado previamente *entropía software*, también ha sido denominada por algunos autores [Neill and Laplante, 2006] como *deudas de diseño*, y en ambos casos, se puede intentar dar solución a través de la aplicación de refactorizaciones [Fowler, 1999].

Aparentemente, el uso del nuevo paradigma orientado a objetos, incluía una serie de propiedades intrínsecas que mejoraban los factores de calidad del software y su producción [Meyer, 1988]. Sin embargo, en la producción utilizando orientación a objetos, se han repetido problemas ya detectados en el mantenimiento y evolución en otros paradigmas. [Bennett and Rajlich, 2000] señalan que incluso aparecen nuevos problemas en el mantenimiento intrínsecos a la orientación a objetos.

[4] Traducción del término inglés *Refactoring*.

En [Brooks, 1995] se describe también el entusiasmo inicial que suscitó dicho paradigma, pero la retroalimentación positiva en el desarrollo no se observa en las primeras iteraciones, sino en iteraciones más avanzadas, en las que las características intrínsecas al paradigma sí que debían facilitar una mejora en el desarrollo al impulsar la reutilización.

Como respuesta a un problema ya conocido, en [Opdyke, 1992, Roberts, 1999b, Fowler, 1999] se elaboran distintas propuestas más o menos formales, para modificar un sistemas software, mejorando su estructura interna, sin alterar su comportamiento, refactorizando.

En dichas propuestas, hay que señalar las siguientes características:

- los cambios se realizan sobre un sistema ya construido, en la línea de la evolución en diseño.
- el sistema funciona correctamente antes y después de aplicar las refactorizaciones.
- los cambios no se realizan con el objetivo de producir cambios en el comportamiento observable del sistema.

Los cambios de los que se trata son los que surgen ante la necesidad de facilitar la comprensión del sistema, su modificación y extensión posterior. No desde el punto de vista de realizar acciones correctoras para corregir un mal funcionamiento del sistema. Como se indica en [Fowler, 1999], el objetivo de refactorizar es *"mejorar el diseño del código después de que se haya escrito"*, pero no cambiar su comportamiento.

Esta evolución y cambio continuo del software ha sido afrontada en metodologías ágiles como en [Beck, 1999, Beck and Andres, 2004]. En este contexto, la evolución del software establece que un programa debe cambiar hacia los nuevos requisitos, en un modo que mejora su diseño en lugar de degradarse. Como ejemplo, eXtreme Programming (XP) es una metodología que adopta, integra y anima al cambio del software, donde uno de los elementos claves de XP es la integración de la refactorización en el proceso de desarrollo como pieza clave del cambio continuo. Por lo tanto adopta la refactorización como *mantenimiento preventivo*, reduciendo los efectos de los futuros cambios.

Es importante diferenciar que cuando se habla de refactorización, se habla de una actividad constante en el proceso de construcción y evolución del software, mientras que cuando se habla de reingeniería se aplica a un sistema software que ha estado en explotación y que por diversas cuestiones debe ser reestructurado para resolver diversos problemas. Además, en reingenieríe el sistema debe ser entendido, pasando a niveles de abstracción diferentes, mientras que en la refactorización se está trabajando siempre en el mismo nivel de abstracción.

Llegado a este punto, siempre cabe la discusión sobre por qué se llega a construir *software* que no es fácilmente modificable, siguiendo un diseño incorrecto o confuso, y que finalmente crea un producto difícil de mantener ante cambios en los requisitos y necesidades inicialmente planteadas. Esta discusión queda fuera del ámbito del presente trabajo.

1.4. Dependencia del lenguaje de programación en refactorización

La mayoría de trabajos en refactorización están vinculados a un lenguaje de programación concreto. Dicha tendencia es difícil de evitar, puesto que la realización de la solución de diseño planteada exige código, y este código siempre está escrito en algún lenguaje particular y concreto, si se quiere aplicar de manera práctica.

Esto es más evidente, al ver el continuo desarrollo de herramientas de refactorización para los distintos lenguajes de programación. En la práctica están orientadas en general a un único lenguaje objetivo. De hecho, éste puede ser un criterio clave a la hora de elegir lenguaje de programación y entornos de desarrollo al aplicar nuevos métodos y procesos [Beck, 1999, Beck and Andres, 2004].

Este problema ya fue detectado en [Opdyke, 1992]. Al establecer refactorizaciones primitivas o de bajo nivel, muchas de ellas sólo eran aplicables a lenguajes orientados a objetos fuertemente tipados, con modificadores de acceso, y que incluyeran conceptos como punteros y aritmética de los mismos. En este caso, el trabajo se centraba en C++, pero al intentar aplicar dichas refactorizaciones a otros lenguajes como SMALLTALK, con características muy diferentes, se pierde la posibilidad práctica de aplicación.

Como se remarca en [Johnson and Opdyke, 1993], *"una refactorización siempre será dependiente del lenguaje, porque debe entender el lenguaje de los programas que está manipulando"*. Aceptando dicha afirmación, hay que resaltar que al analizar y aplicar el mismo tipo de refactorizaciones sobre diferentes lenguajes, se puede observar que el modo en que se llevan a cabo mantiene ciertos puntos comunes. Por lo tanto, existen diferencias, pero también similitudes que pueden y deben ser tenidas en cuenta.

Además, desde el punto de vista de un desarrollo con y para reutilización, es interesante aportar la definición e implementación de refactorizaciones lo más reutilizable posible. Los grandes esfuerzos realizados en el estudio de las refactorizaciones, junto con los métodos, procesos, herramientas, métricas, heurísticas, etc. utilizados en un lenguaje concreto, hacen que sea necesario considerar la posibilidad de reutilizar los resultados obtenidos en un contexto particular en otros contextos más amplios.

1.5. Hacia una cierta independencia del lenguaje

Mens y Tourwé en su conocido trabajo sobre refactorización [Mens and Tourwé, 2004] indicaron como tendencia en investigación la definición de refactorizaciones con independencia del lenguaje. Dicho término debe ser matizado, puesto que hablar de una independencia total del lenguaje no sería del todo correcto, como se ha planteado anteriormente.

Los modelos formales y herramientas deben ser suficientemente abstractos, para poderse aplicar a varios lenguajes, pero también proporcionar los enganches adecuados para las particularidades de cada lenguaje [Mens and Tourwé, 2004].

Incluso utilizando soportes que permitan recoger diferentes elementos sintácticos, la semántica estática y dinámica [Meyer, 1990] de cada lenguaje puede variar en gran medida. Dado que todos estos elementos influyen a la hora de definir una refactorización, sería mejor hablar de reutilización de la definición y de las herramientas de soporte.

La complejidad es más evidente, cuando incluso dentro de un propio lenguaje, pueden existir distintos elementos que dificulten la aplicación de las refactorizaciones. Por ejemplo en JAVA o .NET, surgen nuevas propiedades, basadas en anotaciones o atributos, que pueden afectar o ser afectados por las refactorizaciones ya existentes, o sugerir nuevas oportunidades de refactorización.

Así pues, dentro de la dificultad asociada a enfocar las refactorizaciones desde el punto de vista de la independencia del lenguaje, también es cierto que deben buscarse algunos de los beneficios intrínsecos al incluir ese nivel de abstracción, en particular si aumentan las posibilidades de reutilización y se reduce el esfuerzo.

1.6. Genericidad y refactorización: ¿cuál es la situación actual?

Aunque el concepto de la programación genérica y la genericidad, ha sido estudiado ampliamente [Musser and Stepanov, 1988, Dehnert and Stepanov, 1998] y en distintos ámbitos, los primeros trabajos sobre refactorización no abarcaron dichos campos, quizás motivado por la complejidad de abordar dicho tema en un lenguaje como C++ [Opdyke, 1992], su imposibilidad de aplicación en lenguajes dinámicamente tipados como SMALLTALK [Roberts, 1999b], o la ausencia del concepto en el propio lenguaje JAVA en sus primeras versiones [Fowler, 1999].

El actual empuje que la programación genérica disfruta, como resultado de su inclusión (o extensión de características) en los lenguajes de la corriente principal (e.g. .NET, JAVA), apunta la necesidad de considerar este aspecto en la definición de refactorizaciones.

Este hecho se constata con la aparición de trabajos que abordan refactorizaciones aplicadas sobre programación genérica en la última década (ver Capítulo 2), pero sin tener un reflejo real en los catálogos de refactorizaciones más conocidos, en la herramientas de refactorización o en su integración en los entornos de desarrollo más utilizados en la actualidad. Por lo tanto, parece éste un campo ideal de trabajo en el que se debe investigar la definición y construcción de refactorizaciones en programación genérica.

1.7. Objetivos del trabajo

El interés en obtener una cierta independencia del lenguaje de programación en la refactorización del código y la percepción de los problemas que surgen al incorporar la programación genérica en el desarrollo y mantenimiento de los sistemas software, nos conducen a afrontar la importancia emergente de considerar refactorizaciones centradas en genericidad.

Así pues se propone el estudio de un catálogo de refactorizaciones sobre programación genérica. A partir de dicho estudio se plantea como objetivo general:

La definición y ejecución de refactorizaciones con el mayor grado de reutilización para la familia de lenguajes orientados a objetos, estática o fuertemente tipados, con capacidades para programación genérica.

Para la consecución de dicho objetivo general se pueden establecer los siguientes subobjetivos básicos:

- *Almacenamiento y recuperación de la información de código.* En concreto para lenguajes OO estáticamente tipados, utilizando MOON [Crespo, 2000] (ver Capítulo 3 y Capítulo 6) como solución de base para su reutilización.
- *Definición de los elementos básicos de las refactorizaciones.* Se tomará MOON como marco de trabajo, identificando los puntos de variabilidad necesarios para la aplicación posterior de refactorizaciones sobre lenguajes concretos.
- *Caracterización de las refactorizaciones en base a los elementos que la conforman.* Incluyendo un modelo de caracterización que permita clasificar y establecer relaciones de cara a la implementación y reutilización de las mismas. Desde el punto de vista del constructor de herramientas y del programador como usuario de la herramienta.
- *Catálogo de refactorizaciones en programación genérica.* Se plantea un nuevo conjunto de refactorizaciones centradas en cuestiones relativas a la programación genérica a través de las clases y métodos genéricos soportados en los lenguajes de programación orientados a objetos.
- *Validación de la solución planteada sobre el catálogo de refactorizaciones.* Se revisa la solución planteada en la construcción de las refactorizaciones del nuevo catálogo establecido previamente, validando la solución propuesta.
- *Establecer un proceso de refactorización.* Asegurando en cierta medida la preservación del comportamiento, como resultado de la aplicación de las refactorizaciones definidas.
- *Construcción de un prototipo.* Donde se incluyan los elementos previamente definidos.

1.8. Resultados esperados

Como resultado de la consecución de los objetivos fijados, se pretende conseguir una serie de beneficios, siempre dentro de las dificultades del planteamiento de la búsqueda de una cierta independencia del lenguaje.

Se señalan como beneficios esperados:

1. Base firme que permita **reutilizar** el esfuerzo de definición e implementación de las refactorizaciones sobre una familia amplia de lenguajes (que puedan ser proyectados sobre MOON), sin repetir de nuevo el proceso desde cero.

   a) Aplicado al caso de entornos de desarrollo integrados que incluyan a uno o varios lenguajes.
   b) Aplicado a herramientas de refactorización.
   c) Implementación de las refactorizaciones analizadas con un enfoque **para y con reutilización** de los elementos determinados en las mismas.

2. Ayuda a la ejecución de la refactorización, parcialmente guiada por el programador, asegurando en cierta medida la preservación del comportamiento.

3. Extensión del conocimiento sobre el proceso de refactorización en programación genérica, sus posibles aplicaciones, y problemas a la hora de ser llevado a cabo, campo en el que la experiencia es actualmente limitada.

4. Prototipo operativo que permita llevar a cabo el proceso definido.

1.9. Estructura del documento

En lo que sigue, el documento se estructura de la siguiente forma:

El Capítulo 2 (*Estado del arte*) presenta los conceptos principales en cuanto a mantenimiento del software, evolución y refactorización, así como el estado del arte y trabajos relacionados en estos campos. Por otro lado, se introduce el concepto de tipos paramétricos y clases genéricas, profundizando en la idea de la programación genérica como nuevo campo de trabajo en el mantenimiento, y en particular en la refactorización del software.

El Capítulo 3 (*Independencia del lenguaje en refactorización*) esboza el esquema general del problema desde el punto de vista de la independencia del lenguaje, abordando la evolución desde una gramática a un metamodelo basado en el lenguaje minimal MOON.

El Capítulo 4 (*Definición, caracterización y proceso de construcción de refactorizaciones*) analiza las definiciones existentes de refactorizaciones, estableciendo la estructura de definición utilizada a lo largo del presente trabajo. Se presenta un modelo de caracterización de las refactorizaciones, aplicable en su construcción e integración en herramientas. Finalmente el proceso a aplicar para el refinamiento y construcción de las mismas.

El Capítulo 5 (*Catálogo de refactorizaciones sobre genericidad*) define un catálogo de refactorizaciones vinculadas a las propiedades particulares de las clases genéricas. Se establece el lenguaje de definición a utilizar, y se definen los distintos elementos que forman la refactorizaciones, según la estructura establecida previamente.

El Capítulo 6 (*Solución basada en frameworks*) propone y desarrolla la implementación de los catálogos descritos desde un enfoque basado en reutilización, basándose en el uso de *frameworks*.

El Capítulo 7 (*Caso de estudio*) detalla la aplicación práctica al extender el metamodelo MOON sobre un lenguaje de programación concreto como JAVA. Se aplica la solución basada en *frameworks* para la construcción de refactorizaciones, con y sin genericidad, detallando el proceso completo y la recuperación del código refactorizado.

Se concluye en el Capítulo 8 (*Conclusiones y líneas de trabajo futuro*) con los resultados obtenidos en la elaboración de esta tesis, las conclusiones y líneas de trabajo que quedan abiertas para un futuro.

CAPÍTULO 2

ESTADO DEL ARTE

En este capítulo se presenta el estado del arte actual, en cuanto a dos puntos principalmente. En primer lugar se revisa el concepto de refactorización (ver Sec. 2.1), desde su origen hasta la actualidad, mostrando los catálogos de refactorizaciones generalmente aceptados y su integración en herramientas. También se revisa el trabajo realizado en aspectos menos conocidos, como su caracterización, proceso y enfoque de independencia del lenguaje de programación. Una vez presentados los trabajos relacionados sobre refactorización, se introduce el concepto de genericidad en los lenguajes de programación orientados a objetos (LPOO) estáticamente tipados (ver Sec. 2.2), analizando la situación en el conjunto más representativo. Finalmente se relacionan ambos conceptos mostrando el soporte dado a las refactorizaciones sobre genericidad.

2.1. Refactorización

Las refactorizaciones son bien conocidas y utilizadas en la actualidad. En este apartado se presenta un estudio de su evolución desde su primera definición hasta su uso hoy en día, incluyendo las refactorizaciones en las herramientas de desarrollo más extendidas. Como resultado de este estudio se presentarán aquellos puntos que no han sido suficientemente desarrollados y a los que posteriormente se intentará dar respuesta en el presente trabajo.

2.1.1. Transformación de programas

Un programa es un objeto estructurado con semántica. Entendiendo por **semántica estática** como *"la descripción de restricciones estructurales que no pueden ser adecuadamente capturadas por descripciones sintácticas"* [Meyer, 1990], mientras que la **semántica dinámica** se define como *"la descripción de los efectos de la ejecución del programa"* [Meyer,

1990]. El hecho de que el programa tenga una estructura permite transformarlo siempre en el marco establecido de su gramática, mientras que la semántica proporciona los medios para comparar y razonar sobre la validez de dichas transformaciones.

La transformación de programas ha sido estudiada en diferentes ámbitos. Aunque originalmente se estudió como aproximación a la implementación de intérpretes, compiladores y optimizadores [Aho et al., 1990, Aho et al., 2006].

El término *"transformación de programas"* es también usado para una descripción formal de un algoritmo que implementa transformaciones. Es decir, es el hecho en sí, y la operación. El lenguaje del programa que es transformado, y el del programa resultante, son llamados *lenguaje origen* y *lenguaje destino*, respectivamente.

Puesto que una transformación implica un *cambio* o transmutación hacia algo nuevo, las transformaciones aplicadas sobre los programas modifican su estructura y semántica con el objetivo de mejorar alguna de sus propiedades. Uno de los propósitos de un marco general para la transformación de programas es definir transformaciones que sean reutilizables en un amplio rango de lenguajes.

En el caso particular de los lenguajes de programación, dado que pueden ser agrupados por similitudes estructurales y/o semánticas, sus transformaciones asociadas deberían ser también parcialmente reutilizables. Por ejemplo, el concepto de variable, y ligadura de la variable, es compartido por todos los lenguajes de programación. Transformaciones en las variables como *renombrado*, *sustitución*, *unificación*, etc. pueden ser definidas para varios lenguajes de manera común.

2.1.2. Taxonomía de transformaciones

Se pueden distinguir dos grandes grupos dentro de las transformaciones [Visser, 2001]:

De traducción transforman un programa a *otro* lenguaje. Incluso cambiando el nivel de abstracción, aunque generalmente con cierta pérdida de información.

De reescritura transforman un programa al *mismo* lenguaje (el lenguaje origen y destino son el mismo). Normalmente este proceso de reescritura conlleva algún resultado, como mejoras en el diseño, mayor facilidad de comprensión, optimización en el rendimiento, etc.

Un desglose más detallado de las distintas variaciones de traducción y reescritura lleva a la siguiente clasificación, no completa, ni cerrada:

Traducción

- ***Migración de programas:*** adaptar un programa a otro lenguaje (u otra versión del mismo) en el mismo nivel de abstracción.
- ***Síntesis de programas:*** transformar rebajando el nivel de abstracción del programa origen.

  - Refinamiento de programas: síntesis donde una implementación (eficiente) es derivada de una especificación de alto nivel, satisfaciendo la especificación inicial.
  - Compilación de programas: síntesis en la cual un programa de alto nivel es transformado a código máquina.

- ***Ingeniería inversa:*** extraer de un programa información de mayor nivel.
  - Decompilación: extracción del código de alto nivel a partir de código máquina.
  - Generación de documentación: extracción y elaboración automática de la documentación del código a partir de información contenida en el mismo.
  - Visualización de software: uso de gráficos y animaciones por computadora para ayudar a presentar y comprender programas.
  - Extracción de arquitectura: recuperación de la arquitectura del sistema a partir de la información del código de sistemas legados.

- ***Análisis de programas:*** reduce un programa a un aspecto particular concreto.
  - Análisis del flujo de control: obtención, en forma de grafo, de todos los posibles caminos recorridos en un cierto programa durante su ejecución.
  - Análisis del flujo de datos: obtención de la información del posible conjunto de valores en varios puntos de un programa en relación con sus variables y expresiones contenidas.

Reescritura

- ***Normalización de programas:*** reduce un programa eliminando complejidades sintácticas.
  - Simplificación de programas: un programa es reducido a una forma normal o canónica (estándar) a partir de la cual no se puede simplificar más.
  - Eliminación de "azúcar sintáctico": algunas de las construcciones del lenguaje son eliminadas traduciéndolas en construcciones más básicas.

- ***Optimización de programas:*** transformar mejorando los tiempos de ejecución o reduciendo el tamaño de los recursos necesarios.
  - Especialización de programas: optimizaciones realizadas bajo la observación del rendimiento de un programa ante las entradas concretas y particulares que recibe.
  - Function *"Inlining"*: expansión del cuerpo de una función en todos aquellos puntos donde se utiliza.
  - Fusión: mezcla de código, como por ejemplo la fusión de dos bucles, para optimizar rendimiento.

- ***Ofuscación de programas:*** transformar dificultando la comprensión del código utilizando renombrados ininteligibles, inclusión de código no ejecutable, etc.
- ***Renovación del software:*** renovar el software para arreglar algún defecto previo (o *bug*) detectado en versiones previas. Suele implicar un proceso de reingeniería.
- ***Refactorización de programas:*** mejorar el diseño del programa de tal forma que sea más fácil de comprender y modificar, preservando su comportamiento.

En la Tabla 2.1[1], se representan los distintos tipos de transformaciones de traducción, **siempre entre diferentes lenguajes**, así como las categorías asociadas en su caso. Se pueden observar transformaciones entre lenguajes diferentes al mismo nivel de abstracción manteniendo la misma información inicial (**migración**) y transformaciones sin pérdida de información reduciendo el nivel de abstracción (**síntesis**). Dentro de la **ingeniería inversa** se realizan transformaciones en un mayor nivel de abstracción sin pérdida de información, como en la decompilación, o con una cierta pérdida en mayor o menor medida, como en la documentación, generación de arquitectura o visualización. Finalmente, existen transformaciones que aumentan el nivel de abstracción, siempre con pérdida de información en el proceso (**análisis**).

Tabla 2.1: Resumen de características de transformaciones de traducción

	Pérdida de información	**Nivel de abstracción**
Migración	=	=
Síntesis		
Refinamiento	=	−
Compilación	+	−
Ingeniería inversa		
Decompilación	=	+
Generación de documentación	+	+
Visualización	++	++
Extracción de arquitectura	+	+
Análisis		
Flujo de control	++	++
Flujo de datos	++	++

Por otro lado, las transformaciones de reescritura (ver Tabla 2.2), **siempre se realizan sobre el mismo lenguaje**, pudiendo variar el rendimiento y complejidad (dificultad para comprender el código). Se pueden observar transformaciones reduciendo la complejidad del código (**normalización** y **refactorización**), transformaciones para aumentar la complejidad y posiblemente reduciendo el rendimiento (**ofuscación**), transformaciones para corregir

[1]Símbolo = implica ausencia de cambios, + aumento y − reducción.

básicamente errores en el mantenimiento correctivo (**renovación**) y transformaciones para aumentar el rendimiento (**optimización**).

Tabla 2.2: Resumen de características de transformaciones de reescritura

	Complejidad	**Rendimiento**
Normalización		
Simplificación	$--$	$+$
Eliminación de "azúcar sintáctico"	$-$	$=$
Optimización		
Especialización	$+$	$+$
Inlining	$+$	$+$
Fusión	$+$	$+$
Ofuscación	$++$	$-$
Renovación	$=$	$=$
Refactorización	$--$	$=$

Existen otras taxonomías relativas al ámbito de la evolución del software [Mens et al., 2002] y el cambio en el software [Buckley et al., 2005] más genéricas en su planteamiento que incluyen las transformaciones de programas entre los elementos estudiados, pero sin focalizar en su clasificación, como se ha visto en esta sección. El objetivo de la presente sección es situar claramente el papel de las refactorizaciones en la transformación de programas, con sus diferencias y similitudes con otras transformaciones.

2.1.3. Refactorización: definición y características

En base a la taxonomía previa, entendiendo que las refactorizaciones son una variación de transformación de programas, y más concretamente, una variación de reescritura (ver Apartado 2.1.2), se presentan las diferentes definiciones del concepto de refactorización según los autores más influyentes en este campo, así como las características generales que servirán de guía en este trabajo.

Definición

Según se menciona en [Roberts, 1999b], el término se origina en una cita de Peter Deutch: *"el diseño de interfaces y la fabricación (factoring) funcional constituyen el contenido intelectual clave del software y son mucho más difíciles de crear o recrear que el código"*. Si se entiende por fabricación (*factoring*) esta separación de funciones en objetos, entonces el cambio de las funciones, cuando existen, puede ser llamado *re*factorización (*refactoring*) [Opdyke and Roberts, 1995].

Se tiene la experiencia de que los componentes reutilizables no son diseñados bien a la primera. Pero debido a una constante creación de nuevas aplicaciones es más fácil abstraer

las partes comunes. El motivo principal es que para la mayoría de las personas es más fácil pensar en concreto que en abstracto. Al intentar crear abstracciones, se tiende a generalizar en exceso o se falla en generalizar ciertas partes. Si se examinan aplicaciones concretas, este proceso se realiza de una forma más precisa.

Por lo tanto, las refactorizaciones son aplicadas en procesos iterativos una vez que el software ha sido construido. Este proceso debe ser parcialmente automatizado, puesto que *"un intento de seguimiento manual de las actualizaciones, comprobando que éstas sean consistentes, tendría un alto consumo de tiempo, difícil y propenso a errores"* [Opdyke, 1992].

La definición propuesta en [Opdyke, 1992], fue de las primeras acuñadas. Según el autor, una refactorización es:

> *"Un plan de reorganización que da soporte a un cambio a nivel intermedio. No cambia el comportamiento del programa y mejora su comprensibilidad y facilidad de mantenimiento."*

Como ejemplos de aplicación de refactorizaciones se plantearon tres casos, aunque el autor centró su trabajo en el último punto:

- *Extracción de componentes reutilizables.* Por ejemplo la adaptación a nuevas interfaces de usuario en sistemas legados. La extracción de las interfaces obsoletas requiere un proceso de refactorización.
- *Mejora de la consistencia entre componentes.* Componentes desarrollados por separado pueden tener conceptos comunes que pueden ser abstraídos y generalizados, para una mejor comprensión y mantenimiento.
- *Soporte al diseño iterativo de frameworks orientados a objetos.* En el difícil proceso iterativo de construir un *framework*, se pasa por varias etapas en su diseño, que implican cambios estructurales.

En [Roberts, 1999b] se establece una definición, algo más relajada, en la que la preservación del comportamiento es eliminada. Su razonamiento parte de la complejidad computacional de dicha comprobación. Esto evita la prueba formal de la preservación del comportamiento, utilizando en su lugar el cálculo de predicados de primer orden, algo sobre lo que es más fácil razonar. Esto simplifica el concepto a:

> *"Una transformación que tiene una precondición asociada y que un programa tiene que satisfacer para que la refactorización pueda ser legalmente aplicada."*

Esta definición sigue incluyendo el concepto de refactorización definido en [Opdyke, 1992] pero relajando las condiciones de preservación de la semántica. En particular aboga por el uso de postcondiciones para eliminar análisis posteriores, computando dependencias entre refactorizaciones y usando propiedades obtenidas dinámicamente para llevar a cabo dichas refactorizaciones.

Como propuesta alternativa a la eliminación de análisis estáticos surge otra rama de análisis dinámico donde se propone la comprobación posterior de su corrección sobre los tests. Como debilidad de este sistema, se apunta que funciona bien en tanto en cuanto los tests de los que se disponen sean completos, apoyándose en la cobertura de los mismos. Por otro lado, los propios tests pueden ser modificados como resultado de la refactorización, impidiendo asegurar la preservación del comportamiento.

Finalmente, la definición más ampliamente extendida es la expuesta en [Fowler, 1999], donde se establece una doble visión: refactorización como nombre, es decir como la operación a realizar, y en segundo lugar como acción, es decir como el proceso de aplicación de las refactorizaciones:

> ***Refactorización (nombre)**: "un cambio hecho a la estructura interna del software para facilitar su comprensión y reducir el coste de mantenimiento, sin cambiar su comportamiento observable."*
> ***Refactorización (acción)**: "reestructuración del software, aplicando una serie de refactorizaciones sin cambiar su comportamiento observable."*

Cabe resaltar que de nuevo se incide en la importancia de la preservación del comportamiento, y en facilitar la comprensión del software. No en otras cuestiones, como la optimización del rendimiento, que aunque no modifican el comportamiento observable (o al menos sólo el tiempo de ejecución y el consumo de recursos) tampoco facilitan su evolución posterior en las tareas de mantenimiento.

Características

Las características básicas y fundamentales, que diferencian a las refactorizaciones del resto de transformaciones son:

Incrementar la facilidad de comprensión uno de los principales motivos para refactorizar el código es aumentar su comprensibilidad y facilidad su mantenimiento [Opdyke, 1992, Roberts, 1999b, Fowler, 1999].

Preservación del comportamiento el comportamiento de la aplicación antes y después de la refactorización debe ser exactamente el mismo [Opdyke, 1992, Fowler, 1999].

En el primer punto, no existe apenas discusión entre los distintos autores, respecto a establecer como objetivo fundamental incrementar la comprensibilidad, de manera informal. Sin embargo, hay que señalar la subjetividad de la idea de *"más fácil de comprender"*. La necesidad de establecer criterios objetivos e indicadores es abordada por otros autores, quedando este punto fuera del ámbito del presente trabajo.

Respecto a la propiedad de preservación del comportamiento, existen diferentes visiones. Después de aplicar una refactorización como requisito indispensable, el programa debe ser correcto, sintáctica y semánticamente, desde el punto de vista de la especificación del lenguaje

[Opdyke, 1992]. Una posible línea de actuación sería aplicar la refactorización, salvando previamente el contenido del código fuente a refactorizar, compilar una vez realizada la refactorización, y en caso de detectarse algún error por parte del compilador, volver a la versión previamente guardada.

Sobre esta solución se indican dos problemas:

1. Esta solución puede ser inaceptablemente costosa, especialmente para refactorizaciones de alto nivel con muchas operaciones primitivas y que afectan a gran parte del código.

2. Más importante aún, algunas operaciones podrían alterar el comportamiento del programa y no ser detectadas por el compilador, puesto que la preservación de la corrección sintáctica y la semántica estática respecto al lenguaje, es una condición necesaria pero no suficiente. La preservación de la semántica estática no garantiza la preservación de la semántica dinámica.

Para asegurar la corrección, en cuanto a la preservación de comportamiento, en [Opdyke, 1992] se establecieron siete propiedades o invariantes, a preservar por parte de las refactorizaciones:

1. Única superclase: toda clase sólo tiene una superclase y ésta no puede ser una de sus subclases, pare evitar ciclos en herencia. Aunque su trabajo se centra en C++, sólo considera herencia simple.

2. Nombres de clases distintos: toda clase en el sistema tiene que tener un único identificador. Incluso en presencia de ámbitos anidados o espacios de nombres.

3. Nombre de miembros distintos en una clase. Todas las variables miembro y funciones dentro de una clase tienen distinto nombre.

4. No se redefinen atributos heredados.

5. Signatura compatible en la redefinición de funciones miembro. Se debe señalar la complejidad que esto conlleva al trasladar esta definición a distintos lenguajes, puesto que las reglas de compatibilidad entre tipos pueden ser muy distintas en cada caso.

6. Asignación entre tipos segura: después de una refactorización, el lado izquierdo de una asignación debe ser de igual tipo que el tipo de la variable del lado derecho o bien un supertipo de éste. Esto se aplica también a la relación entre el argumento real y el argumento formal de las funciones.

7. Equivalencia semántica: la equivalencia semántica se define operacionalmente. El programa tiene que producir el mismo resultado para las mismas entradas antes y después de refactorizar.

En [Roberts, 1999b] se indica la gran dificultad de realizar pruebas formales de este último punto, de ahí surge la definición relajada de refactorización que utiliza dicho autor,

y señalada previamente. Por lo tanto, de los puntos anteriores se puede establecer que los seis primeros se corresponden a restricciones de la semántica estática, mientras que el último punto corresponde a la preservación de la semántica dinámica.

Como se señala además en [Mens and Tourwé, 2004], la preservación del comportamiento puede no sólo ligarse a un comportamiento observable en cuanto a las entradas y salidas del programa, sino que puede ser dependiente del dominio del problema o restricciones concretas del usuario como:

- *Software de tiempo real*: donde el comportamiento está vinculado al tiempo de ejecución.
- *Sistemas embebidos*: donde se restringe la memoria y el consumo de energía.
- *Software crítico*: donde propiedades como la fiabilidad y funcionamiento continuo del sistema son fundamentales.

Las refactorizaciones, en la actualidad, se ciñen a una definición de preservación de comportamiento basada en la idea de software de caja negra, con entradas y salidas esperadas, sin consideraciones ligadas a este otro tipo de sistemas con requisitos vinculados a la arquitectura, plataforma, recursos hardware/software o tiempos de ejecución.

2.1.4. Catálogos de refactorizaciones

Una vez establecido el concepto de refactorización y sus características, queda abierta la definición del conjunto que forman los catálogos de refactorizaciones. Dado el carácter práctico de las refactorizaciones, con el objetivo final de incorporarlas en el día a día del mantenimiento del software, se clasifican y agrupan en base a sus propiedades.

Los autores que han definido el concepto, también han definido a su vez sus catálogos de refactorizaciones. En una forma similar a la descripción de los patrones de diseño [Gamma et al., 1995], aunque con cierta disparidad de criterios en cuanto a su definición. A continuación se realiza un estudio de los catálogos más conocidos.

Catálogo de [Opdyke, 1992]

En la tesis doctoral de W.Opdyke [Opdyke, 1992] se identifican y presentan veintitrés refactorizaciones denominadas como "de bajo nivel" (*low-level refactoring*) y el desarrollo de tres refactorizaciones compuestas (*big-level refactoring*).

El catálogo se determina a través de su observación personal sobre las distintas refactorizaciones llevadas a cabo por programadores en lenguajes orientados a objetos. El criterio de agrupación en el catálogo se basa en el concepto de acción y ámbito sobre el que se aplica la acción.

Las refactorizaciones "de bajo nivel" se agrupan utilizando un criterio funcional, exceptuando el último grupo, dada la dificultad de caracterizar esas refactorizaciones por una única operación:

Crear una entidad creación de nuevas entidades de diferente nivel en el programa (3 refactorizaciones).

Borrar una entidad eliminar entidades que no son utilizadas (3 refactorizaciones).

Cambiar una entidad cambiar una entidad, bien sus propiedades o bien su ámbito, en relaciones de cliente y proveedor entre las clases (15 refactorizaciones).

Mover una entidad mover entidades entre diferentes miembros de una jerarquía de herencia (2 refactorizaciones).

Nivel intermedio estas refactorizaciones se clasifican por el autor como compuestas e implica el uso de varias de las refactorizaciones mencionadas previamente (3 refactorizaciones).

Por otro lado, compone tres grandes refactorizaciones a partir del conjunto de refactorizaciones "de bajo nivel": **CREAR UNA SUPERCLASE ABSTRACTA**, **SUBCLASIFICAR Y SIMPLIFICAR CONDICIONALES** y **AGREGACIONES Y COMPONENTES**.

Desde un primer momento ya señala la no aplicabilidad de ciertas refactorizaciones (aquellas dependientes de los tipos) sobre determinados lenguajes, como por ejemplo aquellos que son dinámicamente tipados (*e.g.* SMALLTALK). El catálogo está pensado para su aplicación concreta sobre C++.

Catálogo de [Fowler, 1999]

El autor establece dos niveles a la hora de catalogar sus refactorizaciones:

Refactoring refactorizaciones relativamente más simples con un ámbito bien definido y cuya aplicación puede ser llevada a cabo en un tiempo relativamente pequeño.

Big refactoring refactorizaciones complejas, con un alto coste de tiempo para su aplicación.

Dentro de la primera clasificación, en las refactorizaciones de tamaño pequeño y medio, se establece una segunda clasificación, agrupando las refactorizaciones en función del tipo de situaciones en las que se aplican y los problemas que resuelven:

Componiendo métodos reducción del tamaño de los métodos utilizados (9 refactorizaciones).

Moviendo propiedades entre objetos corrección de las decisiones erróneas de diseño, respecto a qué propiedades deben ir en ciertas clases (8 refactorizaciones).

Organizando datos facilitar el trabajo con los datos (16 refactorizaciones).

Simplificando expresiones condicionales simplificar el uso de condicionales en el código en base al polimorfismo de inclusión (8 refactorizaciones).

Simplificando las invocaciones a métodos mejorar las interfaces utilizadas para la invocación sobre los objetos (15 refactorizaciones).

Manipulando la generalización modificar las jerarquías de herencia actuales (12 refactorizaciones).

Las refactorizaciones en este catálogo se realizan sobre ejemplos escritos en JAVA, con el fin de demostrar su aplicabilidad. En total define 68 refactorizaciones y 4 refactorizaciones grandes (*big refactorings*). La clasificación, al igual que en [Opdyke, 1992], se basa en el concepto de acción y ámbito sobre el que se aplica.

Este catálogo es el que normalmente se toma como referencia básica a la hora de incluir refactorizaciones en las herramientas y los entornos de desarrollo. El conjunto no está cerrado, ya que el propio autor mantiene un repositorio en la web [Fowler, 2013] donde se revisan, actualizan y publican nuevas refactorizaciones.

Para la descripción de las refactorizaciones se utiliza una descripción informal indicando: un nombre, un breve resumen indicando el porqué y qué se realiza, su motivación y razones para no llevarla a cabo, los mecanismos como conjunto de pasos a realizar y un ejemplo demostrativo. Esta plantilla ha sido adoptada como estándar *de facto* por multitud de autores para la descripción de refactorizaciones.

En este catálogo, las refactorizaciones están vinculadas a corregir lo que el autor denomina, *bad smells* o defectos de código. Identifica veintidós defectos que pueden ser detectados en el código, ya sea de forma automática, semiautomática o manualmente y que actúan como indicador para iniciar el proceso de refactorización.

Catálogo de [Kerievsky, 2004]

El autor establece un catálogo de refactorizaciones de alto nivel vinculando el concepto de patrón de diseño y las refactorizaciones. Las refactorizaciones se definen de forma similar a como se realiza en [Fowler, 1999], con alguna pequeña modificación en la plantilla utilizada, y siempre encaminadas a la introducción de un patrón de diseño como parte de la solución final.

El catálogo muestra ejemplos en JAVA. Dadas las similitudes entre los LPOO, se puede extrapolar a los lenguajes de dicha familia, aunque introduciendo variantes en los mecanismos.

Se estructura en seis grupos de refactorizaciones:

Creación aborda los problemas de creación de objetos (6 refactorizaciones).

Simplificación búsqueda de soluciones más simples en métodos, transiciones de estados y estructuras (6 refactorizaciones).

Generalización transformaciones de código de soluciones más específicas hacia soluciones más generales, por lo general eliminando código duplicado y simplificando el código (7 refactorizaciones).

Protección proteger el acceso a propiedades y funcionamiento del programa, con un código más claro (2 refactorizaciones).

Acumulación mejora del código de aquellos objetos que acumulan información, o que la recogen a partir de muchos objetos (2 refactorizaciones).

Utilidades refactorizaciones de bajo nivel necesarias para implementar las anteriores refactorizaciones (3 refactorizaciones).

El autor clasifica a todas las refactorizaciones de alto nivel, excepto aquellas de bajo nivel incluidas en el grupo de utilidades. Sigue la misma nomenclatura que [Fowler, 1999] en cuanto a los dos niveles básicos para clasificar las refactorizaciones.

Las similitudes y dependencias con el trabajo previo de [Fowler, 1999] hacen obligatorio el conocer y dominar dicho catálogo para poder utilizar el catálogo propuesto por Kerievsky.

Catálogo de [Ambler and Sadalage, 2006]

En [Ambler and Sadalage, 2006] se estableció un catálogo de refactorizaciones sobre bases de datos. El punto de encuentro que supuso el auge de los sistemas gestores de bases de datos orientados a objetos, actualmente vigente [Silberschatz et al., 2010, Connolly and Begg, 2009, Elmasri and Navathe, 2010], y el auge del concepto de refactorización en los LPOO, hizo que se extrapolase dicho concepto al mundo de las bases de datos. Para su descripción se sigue usando una plantilla, muy similar a las ya utilizadas en [Fowler, 1999, Kerievsky, 2004].

De nuevo, se establecen grupos por similitudes en el objetivo de la refactorización:

Estructural cambios a la definición de una o más tablas (18 refactorizaciones).

Calidad de datos mejora de la calidad de la información contenida (14 refactorizaciones).

Integridad referencial mejora en las relaciones de integridad referencial al modificar los datos (8 refactorizaciones).

Arquitectura cambios que mejoran la interacción de aplicaciones externas con la base de datos (12 refactorizaciones).

Calidad de métodos cambios a procedimientos almacenados, funciones almacenadas o disparadores, que mejoran su calidad. Dada la similitud con las refactorizaciones sobre métodos [Fowler, 1999] y su alto número, se dividen en dos grupos: cambios en la interfaz (6 refactorizaciones) y cambios internos (11 refactorizaciones).

Transformaciones un cambio al esquema de base de datos que modifica la semántica (5 transformaciones). Propiamente no se clasifican como refactorizaciones, pero se incluyen en el catálogo.

Con excepción del último grupo, el resto sí que incluyen un conjunto de operaciones que podrían clasificarse puramente como refactorizaciones en el contexto de las bases de datos orientadas a objetos.

Otros catálogos

La necesidad de definir catálogos de refactorizaciones es evidente. Dicha necesidad surge en tanto en cuanto el concepto de evolución es intrínseco a cualquier lenguaje de programación, e incluso en otros campos tan diferentes como las bases de datos, por lo que es necesario el concepto de refactorización y catálogos comunes que describan las refactorizaciones, dotando de un vocabulario común para todas las personas implicadas en labores de mantenimiento del software.

Las revisiones a los catálogos y la migración de los mismos hacia otros lenguajes son continuas. [Fields et al., 2009] revisan el catálogo completo de [Fowler, 1999] para un lenguaje como RUBY. Por otro lado también se definen catálogos para lenguajes como FORTRAN [Méndez et al., 2010] en la actualidad. En el campo de la orientación a aspectos también se han hecho propuesta de catálogos de refactorizaciones [Monteiro and Fernandes, 2005].

Como conclusión, es previsible que los catálogos existentes crezcan en cuanto al número de refactorizaciones descritas, y que por otro lado, aparezcan nuevos catálogos en lenguajes y paradigmas de programación para los que todavía no se han establecido.

2.1.5. Refactorización en aplicaciones y entornos de desarrollo integrados

Actualmente, la proliferación de herramientas que asisten a la refactorización del código es un hecho. Tanto desde el punto de vista de aplicaciones independientes como en el desarrollo de *plug-ins* o *add-ins* para su integración en entornos de desarrollo.

Como se establece en [Mens and Tourwé, 2004] para la implantación del uso de refactorizaciones, el soporte de herramientas es crucial, y dichas herramientas deben incluir:

- Un cierto grado de automatización (normalmente semiautomática, con cierta interacción por parte del usuario). Existen propuestas de herramientas que ejecutan de manera automática, sin asistencia del usuario, cambiando el código, pero generalmente en la línea de optimización de código y no con el objetivo de mejorar la estructura y comprensión.
- Fiabilidad en cuanto al resultado de la ejecución de la refactorización, en particular respecto a la preservación del comportamiento.
- Facilidad de configuración y apertura a extensión y modificación de las refactorizaciones. En concreto, la modificación de las refactorizaciones debe poderse realizar de una forma simple, permitiendo conectar las refactorizaciones con la detección de defectos de código por medio de diferentes métodos (métricas, heurísticas, etc.), para finalmente facilitar su composición en la resolución de dichos defectos.

- Cobertura del proceso completo de refactorización, no sólo limitado a la aplicación de las mismas.
- Escalabilidad a la hora de componer refactorizaciones.
- Independencia del lenguaje, pudiendo reutilizar los conceptos sobre un número más amplio de lenguajes.

En las implementaciones actuales, es complicado encontrar una herramienta o *plugin* que cubra todos estos requisitos, pero la tendencia es ir haciendo evolucionar los productos hasta llegar a cumplirlos.

Generalmente se ha tendido a aportar implementaciones de las refactorizaciones del catálogo de [Fowler, 1999]. En la amplia mayoría de herramientas (*plugins* o entornos) se proporciona un menú adicional de opciones, con el conjunto de refactorizaciones implementadas disponibles para ser aplicadas.

El diseño e implementación de las refactorizaciones se abordó inicialmente como simples algoritmos a ejecutar sobre el código, sin tener en cuenta estos criterios antes mencionados en el desarrollo de herramientas de refactorización. Como resultado de esto, criterios como la escalabilidad o la independencia del lenguaje no han sido cubiertos en los primeros desarrollos. Las refactorizaciones implementadas estaban excesivamente acopladas al entorno y lenguaje particular, impidiendo su reutilización y evolución en siguientes versiones, recayendo en los problemas planteados en el propio mantenimiento del software.

El precedente histórico de este tipo de herramientas es *Refactoring Browser* [Roberts, 1999a]. Fue uno de los primeros productos que integraba refactorizaciones en un entorno, en particular sobre SMALLTALK. Soluciones de este tipo se han utilizado en herramientas íntimamente ligadas a un lenguaje particular. Por ejemplo, [RefactorIt, 2007] es un ejemplo de *plugin* que cubre muchos de los criterios señalados, integrado en multitud de entornos, pero limitado al lenguaje JAVA.

En contraposición, un ejemplo claro es ECLIPSE [Daum, 2005]. Su evolución demuestra la importancia de la inclusión en el entorno de las refactorizaciones con el menor impacto posible. En [Frenzel, 2006] se realiza un análisis del concepto de refactorización y el cómo se debe incluir dicho concepto en el entorno. La solución aboga por extraer los elementos recurrentes de estas refactorizaciones, en una capa independiente del lenguaje, habilitando su uso como una interfaz de programación de aplicaciones (API) independiente. Este API se denomina como LTK (*Language Toolkit*), aplicable inicialmente a JAVA. Su aplicación posterior sobre C++, corrobora las posibilidades de reutilización, como prueba del concepto.

En el desarrollo de esta nueva forma de plantear las refactorizaciones, se toma en cuenta que sus efectos pueden ir más allá del propio código fuente. En la actualidad, en un proyecto software, el número de ficheros adicionales en formatos diferentes está aumentando. La evolución que ha sufrido ECLIPSE en los últimos años, hacia un *toolkit* que permite cubrir los criterios de automatización, fácil extensión, escalabilidad, y hasta una cierta independencia del lenguaje, demuestran la tendencia actual.

Diseños como los planteados en las últimas versiones de ECLIPSE, de manera teórica, habían sido recogidos en los trabajos de [Crespo et al., 2004, López et al., 2006b]. En estos trabajos se planteaba la necesidad de establecer una solución basada en *frameworks*, dando una solución de diseño de un motor que permitía la reutilización de los elementos que componen las refactorizaciones en su ejecución.

Esta tendencia se sigue también en la familia de lenguajes de .NET. La naturaleza inherente de la independencia del lenguaje en este paradigma empuja a la construcción de herramientas multilenguaje y, más concretamente, a la aplicación de operaciones (refactorizaciones, por ejemplo) reutilizando esfuerzos previos.

Ejemplos de implementaciones en esta línea, como RESHARPER, se encuentran en [JetBrains, 2013b]. Posiblemente la experiencia ya obtenida por parte de la empresa en el desarrollo de herramientas similares como INTELLIJ IDEA [JetBrains, 2013a], en JAVA, facilitan el diseño e implementación de nuevas herramientas de refactorización. Productos de otros fabricantes como REFACTORPRO [Developer Express Inc., 2013] y VISUAL ASSIST X FOR VISUAL STUDIO [Whole Tomato Software, 2013] siguen esta misma línea, dando implementaciones de su herramienta de refactorización para VB .NET, ASP .NET, C# y C++.

Se puede observar el beneficio de utilizar soluciones enfocadas a varios lenguajes. El desarrollo de herramientas de refactorización para C++ estaba bastante limitado, como por ejemplo con XREFACTORY [Xref-Tech, 2007]. El empuje sufrido al pertenecer ahora a la familia de lenguajes .NET, provoca que los *plugins* de refactorizaciones desarrollados puedan aportar refactorizaciones sobre dicho lenguaje, pese a las dificultades que presenta la implementación de refactorizaciones en C++ [Opdyke, 1992, Johnson and Opdyke, 1993].

Sigue siendo sorprendente, el poco esfuerzo realizado en un entorno convertido en estándar *de facto* para estos lenguajes, como VISUAL STUDIO, en que la inclusión de refactorizaciones está ciertamente limitada, supeditado a la inclusión de *plug-ins* o productos de terceros, como los ya mencionados.

Desde el punto de vista de la reutilización en distintos lenguajes, el ejemplo más claro es X-DEVELOP [Omnicore, 2007b], como entorno integrado para trabajar en JAVA, C#, JSP, J# y VB .NET. La capacidad para trabajar en distintos paradigmas diferencia a este producto de los demás, permitiendo aplicar distintas refactorizaciones, aunque el conjunto actual implementado no sea muy amplio.

Se puede señalar la relevancia actual de las herramientas de refactorización, constatado por la aparición de talleres o *workshops* centrados en la construcción de este tipo de herramientas [Dig, 2007, Dig, 2008, Fuhrer and Opdyke, 2009, Dig and Batory, 2011, Sommerlad, 2012]. Sin embargo, las refactorizaciones en genericidad, no han sido un punto de interés a incluir en herramientas o entornos. Con la excepción de CODEGUIDE [Omnicore, 2007a] e INTELLIJ IDEA [JetBrains, 2013a] que incluyen la refactorización *Generify*, el resto de productos no han centrado su interés en aplicar refactorizaciones sobre estas propiedades.

Su inclusión futura, y el desarrollo de las mismas, teniendo claro los requisitos a cumplir [Mens and Tourwé, 2004], deben conducir al desarrollo de motores y bibliotecas de refacto-

rización reutilizables. Sin embargo, en la mayoría de ejemplos vistos, el escaso soporte a la reutilización y al concepto de genericidad, deja un espacio muy grande a explorar.

2.1.6. Caracterización

Los principales catálogos de refactorizaciones en los que se apoyan las herramientas actuales son los expuestos por Opdyke [Opdyke, 1992] y Fowler [Fowler, 2013]. Ambos autores, en la presentación de sus catálogos, muestran ciertos criterios de agrupación para mejorar la compresión y la organización. Incluso existe solapamiento en algunas refactorizaciones de sus catálogos. También es común a ambos describir motivos o indicios para iniciar la refactorización.

En esta línea, en [Wake, 2003], se amplía el trabajo de Fowler, centrando su trabajo en los defectos de código como guía para la aplicación de refactorizaciones. Aunque en [Fowler, 1999] se estableció una relación entre ambos elementos, Wake centra su trabajo en los defectos como pieza fundamental en contraposición a lo realizado por Fowler. Los defectos son el pivote que dirige el proceso de aplicación de refactorizaciones, usándose para clasificar y agrupar refactorizaciones, dando una visión complementaria del mismo catálogo. Sin embargo, no se aporta realmente una caracterización de las refactorizaciones, sino que se da una visión diferente centrada en la motivación de las refactorizaciones.

Basados en la alta complejidad de este dominio, otros trabajos se enfrentan al problema de clasificación de refactorizaciones y estructuración de su conocimiento. Su objetivo es mejorar la comprensión, unificar conocimiento y enseñar las relaciones existentes entre los diferentes elementos del dominio.

En [Crespo and Marqués, 2001] se presenta una revisión y análisis de refactorizaciones proponiendo una clasificación que permite caracterizar y comparar refactorizaciones definidas en distintos trabajos. Establece siete categorías para caracterizar las refactorizaciones aunque se centran fundamentalmente en el problema de mejorar la comprensión del conocimiento del dominio. Las categorías se enumeran a continuación:

Motivo	Reutilización, cambio de requisitos o mantenimiento y calidad.
Dirección	Horizontal o vertical.
Resultado	Universal o particular.
Consecuencias	Agresiva, respetuosa con los clientes o respetuosa con los objetos.
Método	Inferencia o demanda.
Intervención humana	Manual, semiautomática o automática.
Objeto	Elementos de análisis, de diseño o de implementación.

Algunas conclusiones pueden ser extraídas de este trabajo, que ayuda a inferir propiedades de las refactorizaciones. Sin embargo, este trabajo está realizado con un punto de vista de alto nivel, no aplicable para resolver problemas de implementación de refactorizaciones a más bajo nivel.

En [Zannier and Maurer, 2003] se enfrenta al problema de la definición de una herramienta para refactorizar hacia patrones de diseño. Propone una clasificación basada en tres criterios de complejidad. El objetivo de esta clasificación es asistir al desarrollo de la herramienta propuesta. Se proponen tres categorías: **atómicas**, **secuenciales** y **complejas**. Los conceptos básicos en OO como paquete, clase, método o atributo son usados para definir la propiedad de ámbito. Estas nuevas categorías organizan las refactorizaciones en base a la incidencia/alcance de las transformaciones y por el conocimiento requerido para introducir cambios.

Esta aproximación tiene algunas ventajas: es fácil de aplicar y no depende del detalle de implementación utilizado, además puede ser aplicada a diferentes contextos. Pero por otro lado, como resultado de su aplicación sólo clasifica las refactorizaciones de [Fowler, 1999] en tres grandes grupos, sin diferenciar a un nivel de grano más fino entre las refactorizaciones de cada grupo.

Sobre la base del trabajo en [Crespo and Marqués, 2001], se propone un modelo de caracterización [López et al., 2006a] (ver Fig. 2.1) con el propósito de ayudar en la construcción de refactorizaciones, sustentando sobre los siguientes conceptos.

Requisitos de conocimiento	Requiere patrones de diseño.
Objetivo	Diseño o implementación.
Entradas	Sistema, clase, atributo, método, instrucción y entidad.
Precondiciones	Sistema, clase, atributo, método, instrucción y entidad.
Acciones	Creación, actualización o borrado.
Postcondiciones	Sistema, clase, atributo, método, instrucción y entidad.

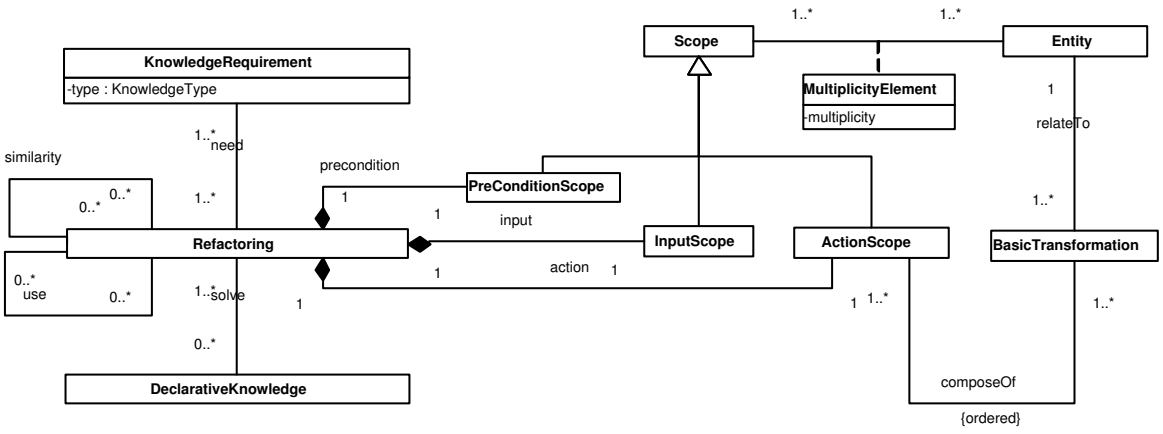

Figura 2.1: Modelo de caracterización de refactorizaciones

Este trabajo toma detalles de más bajo nivel, respecto de [Crespo and Marqués, 2001], como el número de tipos de los elementos usados en las precondiciones, acciones y postcondiciones. En la práctica, se ha detectado que esta solución está ligada a cómo se implementan las refactorizaciones, en la solución particular elegida, dificultando su reutilización en otros contextos. Disponer de toda esa información es útil, pero ciertas decisiones tienen que ser tomadas sin conocer todos lo detalles previamente, lo que dificulta su aplicabilidad. Cambios en la implementación podrían afectar a la caracterización, y este efecto no es deseable.

Por lo tanto, en vista de lo anterior, es necesaria la definición de una caracterización que simplifique su aplicación, pero que por otro lado guíe en la toma de decisiones a la hora de desarrollar herramientas de refactorización o IDEs. Parece necesario establecer unos criterios de caracterización considerando además su uso en un proceso completo de refactorización. A través de éstos, se pueden seleccionar conjuntos de refactorizaciones relacionadas, así como un orden y precedencia de las mismas. Para ello surge como necesidad el uso de modelos de caracterización abiertos para cualificar y cuantificar la complejidad de las refactorizaciones, incluidas en los distintos catálogos, como se detalla en el Capítulo 4.

2.1.7. Proceso

En la construcción de refactorizaciones, se debe establecer de manera clara un proceso. En [Opdyke, 1992], se ensamblan como un conjunto de precondiciones y acciones a ejecutar en un cierto orden. Sin embargo esta solución parte de la idea de que la definición de la refactorización y de los elementos que la componen es precisa *a priori*.

Otras cuestiones como las distintas variaciones a realizar a la definición de la refactorización (y sus elementos asociados) cuando se aplican sobre otros lenguajes, no son consideradas. La misma línea de trabajo se continúa en [Roberts, 1999b], pero con un mayor interés en las postcondiciones, y la composición de las refactorizaciones, en particular para SMALLTALK.

Ambos enfoques son pesimistas en el sentido de que pueden impedir ejecutar refactorizaciones que podrían ser correctas. La principal dificultad estriba en definir el conjunto de precondiciones preciso que asegure una ejecución de la refactorización, que preserve el comportamiento, y que no impida ejecutar la refactorización, en casos en los que podría llegar a ser correcto.

Aún utilizando el mismo tipo de definición y estructura para las refactorizaciones, distintos autores establecen precondiciones distintas para la misma refactorización. Como ejemplo se puede tomar una refactorización relativamente simple como **ELIMINAR CLASE**. Inicialmente cada autor la denomina de forma ligeramente diferente, y además define diferentes precondiciones que impiden su ejecución. Esto revela la complejidad de un establecimiento correcto *a priori* de las precondiciones.

Por ejemplo:

- En [Opdyke, 1992], la refactorización se denomina **DELETE_UNREFERENCED_CLASS** y establece como precondición que la clase no es referenciada y no tiene descendientes.

- En [Roberts, 1999b], la refactorización se denomina **RemoveClass** y establece dos precondiciones que permiten ejecutar la refactorización: la clase no es referenciada y no tiene descendientes, o bien la clase no es referenciada, tiene descendientes pero no tiene métodos o variables de instancia. En este segundo caso, las subclases pasarían a serlo de la superclase de la clase eliminada.

La influencia del lenguaje de programación para el que se plantea la refactorización (C++ y Smalltalk respectivamente) motiva estas divergencias.

En [Fowler, 1999] se proponen definiciones más informales, al estilo de otras soluciones adoptadas en la definición de patrones de diseño, dando una lista de elementos que completan la descripción de una refactorización como una receta de cocina. El proceso de construcción de la refactorización parte de suposiciones, intuiciones y heurísticas a la hora de establecer los elementos de la misma. Este enfoque es más optimista, enfrentándose posteriormente al problema de errores en compilación o en la ejecución de la batería de pruebas.

Aunque muchos autores han trabajado en la elaboración de catálogos (ver Apartado 2.1.4) caracterizados de manera más o menos completa, no se establece claramente un proceso ordenado de definición, implementación y prueba de las refactorizaciones de dichos catálogos.

El propio proceso de definición, elaboración y construcción de las refactorizaciones y herramientas que las soportan está abierto, existiendo cierto desacuerdo entre las principales corrientes. Las características de una refactorización varían entre distintos autores y son dependientes del lenguaje sobre el que se aplica, dándose una solución poco reutilizable y con gran dificultad para la corrección de errores.

Así pues, es necesario establecer un proceso más formal a la hora de construir los catálogos de refactorizaciones, guiados por modelos de caracterización, que asistan a su construcción de una manera ordenada y objetiva. En el Capítulo 4 se muestra el proceso incremental propuesto para la definición de refactorizaciones.

2.1.8. Independencia del lenguaje en refactorización

La independencia del lenguaje ha sido abordada en diferentes trabajos, siempre con el objetivo de aplicar un gran número de refactorizaciones común al mayor conjunto de lenguajes posible.

En [Lämmel, 2002], se introdujo la noción de refactorización genérica de programas, como propuesta inicial hacia un *framework* parametrizable para cada lenguaje. Se sugirió su implementación con Haskell, proporcionando puntos de enganche para los elementos específicos de cada lenguaje. El *framework* podía ser instanciado para diferentes lenguajes como Java, Prolog, Haskell y XML. Como el propio autor apuntó, se trataba de un esquema inicial apuntando los conceptos fundamentales que se deben incluir y parametrizar, tomando como caso de estudio la extracción de abstracciones con el *framework*. Los conceptos manejados eran: interfaces, pasos a realizar, tratamiento, navegación en árboles de sintaxis, análisis genéricos para observar efectos laterales y parámetros específicos del lenguaje. En la medida de nuestro conocimiento no se dispone de una implementación concreta para su validación.

Otras aproximaciones, menos ambiciosas a la hora de la resolución del problema de la independencia del lenguaje están basadas en metamodelos. En [Demeyer et al., 1999, Tichelaar et al., 2000], se define el metamodelo FAMIX para almacenar información en varias herramientas CASE para reingeniería. Este metamodelo especifica las entidades principales en orientación a objeto: clases, métodos, atributos, herencia, etc. Para poder completar el proceso de reingeniería, introducen dos entidades adicionales: la invocación y el acceso a propiedades. Una invocación representa una llamada a método, mientras que un acceso representa el uso de un atributo desde el cuerpo de un método. Estas abstracciones son necesarias para el análisis de dependencias, cómputo de métricas y operaciones de reingeniería. Sin embargo, no soporta el concepto de herencia avanzada (modificación de propiedades en las cláusulas de herencia, herencia múltiple, etc) ni los conceptos relacionados con la genericidad. A partir de dicho modelo se desarrolla un vasto conjunto de herramientas [Ducasse et al., 2000, Ducasse et al., 2001, Nierstrasz et al., 2005] para la recolección de métricas, consulta, navegación, análisis y refactorización.

Siguiendo la base de FAMIX, en [Ducasse et al., 2000, Ducasse et al., 2001, Nierstrasz et al., 2005] se expone el entorno de reingeniería denominado Moose (ver Fig. 2.2). Dicho entorno nace con el objetivo de soportar el proceso completo de reingeniería sobre aplicaciones desarrolladas en diferentes lenguajes. La independencia del lenguaje se enmarca como característica principal pero permitiendo su extensión a los lenguajes particulares. El entorno se basa en FAMIX, en particular en su versión 2.0, aunque en la actualidad están trabajando sobre una versión 3.0 ante la diversidad de variaciones que se ha producido sobre dicho metamodelo.

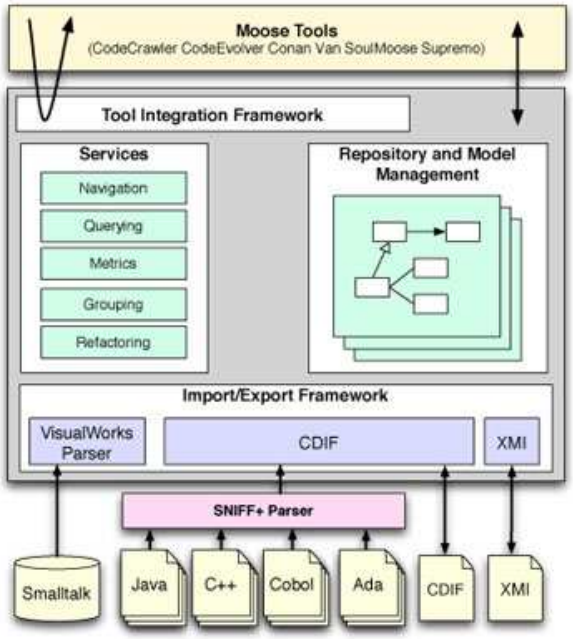

Figura 2.2: Arquitectura de Moose [Ducasse et al., 2001]

El concepto de repositorio MOOSE encaja con la idea de instancias del modelo de FAMIX recogidas del código fuente. Esta información es extraída a partir de *front-end* propios del lenguaje como en SMALLTALK, o bien analizadores generados para el lenguaje particular, como en el caso de JAVA. Hay que señalar que la información extraída es el subconjunto mínimo necesario para poder realizar los posteriores análisis y establecer las condiciones para poder llevar a cabo las refactorizaciones. Información como el cuerpo de los métodos es parcialmente obtenida en el proceso, por lo que las refactorizaciones dependientes de dicha información no pueden llevarse a cabo.

En [Tichelaar, 2001] se describe el motor de refactorizaciones utilizado detalladamente, indicando dos puntos claves:

- el análisis de las precondiciones necesarias para la ejecución de las refactorizaciones se puede llevar a cabo sobre la información del repositorio (o alguna extensión realizada).
- las modificaciones sobre el código se realizan de manera específica, dependiendo del lenguaje analizado. Como ejemplos concretos, en SMALLTALK se realizan con la herramienta SMALLTALK REFACTORING BROWSER [Roberts, 1999a], mientras que en JAVA se realizan con mecanismos basados en edición de textos sobre los ficheros fuente iniciales.

Aunque las precondiciones, no todas, puedan llegar a ser expresadas de manera independiente del lenguaje, finalmente las transformaciones se llevan a cabo bajo el concepto de frontales de transformación de código. Dichos frontales trabajan directamente sobre el código fuente y por lo tanto son específicos del mismo. En este aspecto, la parte operacional es difícilmente reutilizable.

Si se toma como ejemplo la clase `RenameClassTransformer` de MOOSE, correspondiente a la refactorización **RENAME CLASS**, se observan una serie de métodos:

- `changeClassName`
- `changeSuperClassReferenceOf`
- `changeClassMethodInvocation`
- `changeClassAttributeAccessOf`
- `changeTypeOfVariable`
- `changeTypeOfParameter`
- `changeReturnTypeOfMethod`

Mientras que en la implementación en JAVA se implementan todos los métodos, en la implementación en SMALLTALK los últimos cuatro métodos no se implementan. Si la refactorización se aplica sobre nuevos lenguajes, que necesiten operaciones adicionales de cambio,

se pueden plantear problemas en el futuro obligando a cambiar la interfaz de la refactorización/transformación y de las correspondientes clases descendientes para cada lenguaje.

Como se puede observar además, aunque el motor es operativo, el diseño está prefijado al análisis previo, sufriendo en exceso ante la detección de nuevas características en la refactorización o adición de nuevos lenguajes.

Aunque se mencionan diferentes *plug-ins* para trabajar con lenguajes como C++ [Bär, 1999], Java [Tichelaar, 1999], etc, sin embargo estos *plug-ins* están planteados como extensiones de FAMIX, no desde el punto de vista de herencia entre clases, sino como modificaciones de las clases originales, sustituyendo una implementación inicial de FAMIX por la del *plug-in* correspondiente. En este sentido, no se trabaja con soluciones basadas en *frameworks* e inversión de control, sino en una implementación basada en bibliotecas. La evolución en paralelo de todos los *plug-ins* se plantea difícil de llevar a cabo, pudiendo llevar a soluciones divergentes, con alto impacto en el mantenimiento de los diferentes *plug-ins*.

Otro punto a resaltar es que tanto el análisis de la refactorización (precondiciones) y las transformaciones correspondientes están disociadas, trabajando en diferentes niveles, complicando el diseño de la solución propuesta. El uso de diferentes soluciones para cada lenguaje obviamente es un impedimento para la reutilización.

En esta misma línea de FAMIX, una solución basada en metamodelos es propuesta en [Van Gorp et al., 2003b, Van Gorp et al., 2003a]. A través de la extensión del metamodelo de UML 1.4 con ocho modificaciones adicionales independientes del lenguaje, se crea un nuevo metamodelo denominado GrammyUML. El objetivo es resolver las inconsistencias entre las refactorizaciones realizadas en diseño y sobre código. Se definen las refactorizaciones con OCL, validando la solución sobre dos refactorizaciones (**Pull Up Method** y **Extract Method**), de forma teórica. La propuesta no ha sido renovada en posteriores trabajos, y en la medida de nuestro conocimiento, no existen implementaciones actuales.

Otra propuesta similar es la basada en MOON (Minimal Object-Oriented Notation) [Crespo, 2000]. MOON representa las abstracciones mínimas necesarias en la definición y análisis de las refactorizaciones, de modo similar a FAMIX o GrammyUML. Se centra en representar las abstracciones de la familia de lenguajes orientados a objetos, estáticamente tipados, con o sin genericidad. Como principales puntos fuertes, está la inclusión de conceptos de herencia avanzada y un completo soporte a la genericidad y sus variantes de acotación. El objetivo es poner una base sólida definiendo *frameworks* que permitan la reutilización en el desarrollo y adaptación de herramientas de refactorización, tal y como se describe también en [Crespo et al., 2006a]. En siguientes capítulos se realiza una descripción más detallada de esta solución.

Otros trabajos como [Mendoça et al., 2004], en la línea de la independencia del lenguaje, buscan un alto grado de independencia del esquema y tecnología empleada a través de XML. El fundamento básico de su propuesta es utilizar XML como mecanismo de transformación. Sin embargo, pese a lo atractivo de utilizar un metalenguaje de uso general como XML, las consultas y actualizaciones que conforman una refactorización deben ser reescritas para cada nuevo lenguaje de programación concreto, limitando las posibilidades de reutilización.

Concluyendo, las soluciones basadas en metamodelos pueden dar una respuesta a la definición y aplicación de refactorizaciones con cierto grado de independiente del lenguaje, pero todavía es necesario desarrollar más estas propuestas para demostrar su aplicación práctica en el proceso de refactorización.

2.2. Genericidad en LPOO estáticamente tipados

En esta sección se introduce el concepto teórico de tipo paramétrico y acotación, para posteriormente presentar el concepto de programación genérica. Los LPOO han ido evolucionando a lo largo de las dos últimas décadas incrementado el soporte a dicha programación. Se detalla la situación concreta de los LPOO estáticamente tipados, centrándonos en aquellos que pueden ser considerados de la corriente principal, y se finaliza la sección analizando la situación actual de la integración de refactorizaciones en relación a las clases y métodos genéricos.

2.2.1. Tipos paramétricos y acotación de tipo

Definición: *se denomina tipo paramétrico o genérico a un tipo cuya especificación está definida en función de alguna variable de tipo, libre para ser instanciada.*

Un tipo paramétrico es un tipo que describe a otros tipos, siendo a su vez una descripción de objetos. Describe tipos que tienen en común estructura y comportamiento, pero que se diferencian en el tipo de algunas de sus propiedades. Una sustitución de las variables de tipo de un tipo paramétrico instancia un nuevo tipo.

La especificación de tipos paramétricos se corresponde con lo que se denomina cuantificación universal porque la descripción del tipo está expresada en esos términos, *i.e.* la expresión del tipo (ET) es $\forall(t_1 \dots t_n) : ET(t_1 \dots t_n)$. La presencia del cuantificador universal, al igual que en el cálculo de predicados frente al cálculo proposicional, hace del sistema de tipos un sistema de tipos de orden superior.

Un sistema polimórfico permite que un objeto pertenezca a varios tipos [Cardelli and Wegner, 1985]. Se denomina polimorfismo a la capacidad de un elemento de software, por ejemplo una entidad o una expresión, de denotar objetos de más de un tipo posible [Meyer, 1997].

En [Cardelli and Wegner, 1985] se clasifica el polimorfismo en dos grandes grupos, polimorfismo universal y polimorfismo *ad-hoc*:

- El término universal significa que un elemento de software definido para un tipo dado, está definido también para un conjunto potencialmente infinito de tipos.
- El término *ad-hoc* significa que un elemento de software puede estar definido para más de un tipo, refiriéndose a una clase de polimorfismo que se utiliza en ciertos lenguajes de programación como un atajo, como por ejemplo la sobrecarga de nombres de funciones u operadores.

Dentro del polimorfismo universal se encuadra tanto el polimorfismo de inclusión como el paramétrico. El polimorfismo de inclusión se establece por la relación de subtipo. Este polimorfismo tiene una forma dinámica puesto que depende de la ligadura en tiempo de ejecución. Por otro lado, el polimorfismo paramétrico es una forma estática, frente a la dinámica en el polimorfismo de inclusión. Aunque de igual modo, ambos son mecanismos potentes de cara a la reutilización. Sin embargo, el hecho de no especificar o delimitar de alguna forma las posibles sustituciones válidas para las variables de tipo, puede provocar a su vez algún problema. El problema surge de la propia universalidad del polimorfismo paramétrico. Ante dicha situación surgen distintas opciones:

- con las entidades de tipo paramétricos, sólo se pueden utilizar propiedades comunes a todos los tipos, garantizando que cualquier sustitución es válida.
- comprobar que las propiedades utilizadas existen en el tipo sustitución, generando un grave problema de comprobación ante la necesidad de disponer del cuerpo de la clase genérica que implementa el tipo paramétrico.
- como solución intermedia, utilizar tipos paramétricos acotados, indicando condiciones a las sustituciones de las variables de tipo.

Esta última solución intermedia, la acotación de tipo, permite la comprobación del tipo paramétrico y de sus clientes de forma separada. Ha sido la solución adoptada en la amplia mayoría de LPOO estáticamente tipados[2].

Existen distintas variaciones de acotación en sistemas de tipos como se detalla en [Crespo, 2000]. A continuación se da una breve descripción de las distintas variantes de acotación:

Subtipado consiste en indicar un tipo asociado al parámetro genérico formal. Todo parámetro real deber ser un subtipo del tipo utilizado como acotación (combinación del polimorfismo paramétrico con el polimorfismo de inclusión).

Acotación F extensión de la acotación por subtipado que permite una acotación recursiva donde la variable de tipo está acotada por un tipo a su vez definido por dicha variable de tipo.

Emparejamiento basada en la relación de emparejamiento por estructura y signatura entre tipos, que no implica polimorfismo de inclusión.

Cláusulas tal que consiste en asociar las signaturas de las propiedades que debe tener toda sustitución real a la variable de tipo.

Sin acotación los parámetros formales no indican tipos, sino operaciones, utilizando parámetros procedurales.

Para una explicación más detallada de estas variantes se puede consultar [Crespo, 2000, Capítulo 3]. De todas las variantes, la acotación por subtipado y cláusulas *tal que* son las más extendidas entre los lenguajes actuales, como se puede ver en la Apartado 2.2.3.

[2]Con la excepción notable de C++.

2.2.2. Clases y programación genérica

Las clases genéricas son clases que se definen en función de uno o más parámetros de tipo, que se denominan parámetros genéricos formales [Meyer, 1997]. Estos parámetros se utilizan en las signaturas y en el cuerpo de la clase genérica como anotaciones de tipo. Las clases genéricas implementan el concepto de tipo paramétrico en un LPOO [Crespo, 2000], como construcción lingüística.

No todos los LPOO soportan la definición de clases genéricas, sobretodo si se tiene en cuenta que sólo tiene sentido en lenguajes estáticamente tipados, lo que descarta la familia de lenguajes dinámicos, como por ejemplo SMALLTALK [Goldberg and Robson, 1983].

Un estudio del soporte de clases genéricas, sobre lenguajes académicos, se puede consultar en [Crespo, 2000]. Sin embargo, en el presente trabajo se establecerá un estudio sobre los lenguajes más extendidos dentro de la comunidad de desarrolladores.

La programación genérica utiliza estos módulos anteriormente descritos, las clases genéricas, con parámetros definidos sobre una interfaz bien general. Ésta describe exactamente qué propiedades se requieren en un entorno, para que el módulo funcione correctamente. Básicamente se busca una reutilización fiable con una definición lo más general posible.

Las primeras apariciones de la programación genérica aparecen a finales de los setenta, con lenguajes como CLU [Liskov, 1976, Liskov et al., 1981] y ADA [Hutchison et al., 1981], para posteriormente ser adoptada en varios lenguajes orientados a objetos, como BETA [Madsen et al., 1993], C++ [Stroustrup, 1997], EIFFEL [Meyer, 1992] o D [Bright, 2004].

En [Musser and Stepanov, 1998a] se da una definición más precisa de la disciplina de la programación genérica: *es el área que trata de encontrar representaciones genéricas de algoritmos eficientes, estructuras de datos y otros conceptos de software y su organización sistemática*. El objetivo es expresar algoritmos y estructuras de datos en una forma ampliamente adaptable que permita su uso directo en la construcción del software.

Una definición más sencilla se encuentra en [Stroustrup, 1997], donde se referencia a la programación genérica como *la programación que utiliza tipos como parámetros*, coincidiendo con las ideas planteadas anteriormente.

Una buena aproximación práctica a la programación genérica utilizando iteradores fue promovida por la *Standard Template Library* (STL), parte de la biblioteca estándar de C++ en la actualidad. En [Musser and Stepanov, 1998b] se señalan otros proyectos interesantes en programación genérica.

2.2.3. Integración de la genericidad en lenguajes de programación orientados a objetos

La programación genérica está ampliamente extendida en los LPOO estáticamente tipados. Algunos lenguajes, como EIFFEL [Meyer, 1997] y C++ [Stroustrup, 1997], incluyeron esta característica desde sus especificaciones iniciales, en mayor o menor medida, mientras que

otros lenguajes la incorporaron posteriormente (*e.g.* JAVA [Bracha et al., 2001] o C# [ECMA, 2006]).

Hoy en día, los lenguajes denominados "de la corriente principal" *(mainstream languages)* incluyen en su mayoría propiedades relacionadas con la genericidad (o se prevé su inclusión). Sin embargo cada lenguaje aporta diferentes soluciones desde un punto de vista sintáctico y semántico. Por ejemplo, JAVA añade la noción de "tipo desconocido", .NET incluye la opción de añadir la signatura del constructor sin argumentos como restricción (ECMA-89 y ECMA-334) [ECMA, 2006, ECMA, 2012], etc.

En [Garcia et al., 2007] se realiza una comparativa más completa entre varios lenguajes (no sólo LPOO estáticamente tipados) sobre sus propiedades en genericidad, pero debemos señalar que desde su publicación hasta la fecha actual, muchos de los LPOO han incluido cambios de profundo calado, que deben ser tenidos en cuenta.

En los siguientes apartados se detalla cómo se ha incluido el concepto de genericidad y programación genérica en los lenguajes de la corriente principal. No se entra en detalles de bajo nivel de implementación por parte del compilador, sino que se centra el estudio en el punto de vista de lenguaje de programación y sistema de tipos.

Para la selección de los lenguajes objeto de estudio se ha tomado como referencia el índice TIOBE [TIOBE Software BV, 2013], *webs* con índices de popularidad como [DedaSys, 2011] o herramientas de recopilación de estadísticas de uso como [Montmollin, 2013]. En todas ellas, nos encontramos con C++, C#[3] y JAVA entre los diez lenguajes de programación más utilizados, siendo los máximos representantes de los LPOO estáticamente tipados.

En el estudio se ha incluido EIFFEL, que si bien no está tan bien posicionado en las estadísticas de uso, ha sido un lenguaje de referencia en genericidad, siendo de los primeros lenguajes en incorporar la programación genérica como elemento fundamental. El trabajo previo de partida [Crespo, 2000] ya tomó EIFFEL como lenguaje de referencia para demostrar la validez de su propuesta bajo las mismas consideraciones.

Genericidad en C++

Evolución histórica Las plantillas se introducen en C++ por motivos similares a la inclusión de genericidad en otros lenguajes, proporcionando un medio de desarrollar contenedores seguros en cuanto a tipos y posibilitar el uso elegante de dichos contenedores (programación genérica). El objetivo era reducir el uso de macros y conversiones explícitas de tipos (moldeados o *cast*). Las plantillas en C++ están inspiradas parcialmente por los genéricos de ADA y los módulos parametrizados de CLU.

Aunque las plantillas de funciones no son necesarias para el desarrollo de contenedores polimórficos seguros en cuanto a tipos, C++ soporta plantillas de clases y de funciones indistintamente.

Desde el primer diseño, no se incluyó la posibilidad de restringir o limitar las sustituciones de los parámetros de la plantilla explícitamente. Aunque fueron realizadas varias propuestas,

[3]Se toma C# como lenguaje referencia para la plataforma .NET.

incluyendo acotación por subtipado, se determinó que dichas propuestas limitaban la expresividad, o eran inadecuadas para expresar la variedad de acotaciones que podrían necesitarse en la práctica [Stroustrup, 1994]. Dicha limitación se ha mantenido en la última especificación [Becker, 2011]. Se habla de propuestas de futuro que introducen la noción de *concept* [Gregor et al., 2006] como soporte a la genericidad restringida, pero no han sido incluidas en la propuesta actual.

Soporte actual El compilador C++ opta por realizar una sustitución en tiempo de compilación de la variable de tipo por aquellas sustituciones que se producen. De forma que, por ejemplo, a partir de `List<String>` y `List<int>` se generan dos tipos distintos en fase de compilación. El proceso es similar al que se produciría con una sustitución manual de la variable de tipo por sus correspondientes sustituciones.

Las comprobaciones de corrección en cuanto a tipos, se realizan a posteriori, una vez realizada la fase de sustitución. Por lo tanto no se necesitan mecanismos de restricción para indicar qué sustituciones son válidas. Sin embargo, esto provoca que el código se vea inflado por la aparición de tantas clases como instanciaciones distintas de la plantilla se produzcan. También impide la ocultación de implementación, ya que las plantillas deben quedar enteramente programadas en los archivos de cabecera (.h).

Sin embargo, se permiten varias operaciones que no se aplican en genericidad a otros lenguajes [Golding, 2005, Becker, 2011]:

- especializar la plantilla en función del tipo sustitución.
- extender de los parámetros de tipos (dentro de la idea de los *mixin*).
- utilizar constantes como sustituciones de tipo.
- plantillas con número variables de parámetros genéricos (*variadic template*) .
- definición de alias de plantillas.

Terminología En C++ se utiliza la siguiente terminología:

Plantillas	También denominadas plantillas de clase, permiten el uso de tipos como parámetros en la definición de una clase o función.
Funciones plantilla	Funciones parametrizables por tipos. Los tipos de los argumentos pasados a la función determinan qué versión es utilizada, en contraposición a las plantillas de clase. Se permite la indicación explícita de tipos para resolver conflictos. Se permite su sobrecarga.
Parámetros de plantilla	Parámetros libres de tipo declarados en la plantilla para ser sustituidos por parámetros de tipo, tipos ordinarios (Ej: `int`) y recursivamente, parámetros de plantilla. Una plantilla puede tener varios parámetros de plantilla.
Instanciación de plantilla	Proceso de generar una declaración de clase o función a partir de una plantilla y sus argumentos.
Argumento de plantilla	Argumentos de tipo utilizados en la instanciación de la plantilla. Pueden ser valores contantes, la dirección de un objeto o función, o un puntero a miembro no sobrecargado.
Especialización	Versión de una plantilla para un argumento de plantilla especial. En general se utiliza para evitar la explosión de código resultante, y compartir de una manera más óptima el código.
Plantillas miembro	Cuando en una clase, o plantilla de clase, se tienen a su vez miembros que son también plantillas.

Variantes de acotación En las plantillas en C++, no se indica ninguna restricción explícita. Es el compilador, el que una vez realizada la instanciación (sustitución), comprueba la corrección de las invocaciones sobre las variables cuyo tipo coincide con el parámetro de tipo. En caso de necesitar la aplicación de alguna rutina común, problema habitual al requerir restricciones sobre las posibles sustituciones, se utilizan argumentos adicionales de la plantilla para proporcionar las operaciones necesarias. Clasificando el conjunto de restricciones de C++, se señalaría que no existe acotación explícita en los parámetros de tipo.

Subtipado en genericidad Como se recoge en [Stroustrup, 1997], no existe relación entre dos clases generadas desde una misma plantilla. Un ejemplo típico es la no relación entre un `Conjunto<Figura>` y un `Conjunto<Círculo>` pese a que exista una relación de herencia (subtipado nominal) entre las clases que son el parámetro real del tipo. Puesto que una posterior asignación permitiría manejar los conjuntos de círculos como conjuntos de figuras, y con esto se podría intentar introducir un triángulo en un conjunto de figuras que realmente maneja un conjunto de círculos, siendo imposible su tratamiento como círculo. Se permite establecer ciertas relaciones entre clases a partir del uso de plantillas miembro [Stroustrup, 1997].

Genericidad en C#

Evolución histórica En [Kennedy and Syme, 2001], los autores muestran una primera propuesta de diseño e implementación de clases genéricas en .NET tomando como lenguaje

de referencia C#. Se permite el concepto de genericidad en las clases, interfaces y estructuras, se trabaja con métodos genéricos, acotaciones y acotación F.

La inclusión real de la genericidad como característica del *framework* .NET se produce a partir de la versión 2.0 del mismo. Anteriormente se utilizaban mecanismos de polimorfismo de inclusión como alternativa, con todos los problemas intrínsecos asociados: falta de seguridad de tipos, utilización de moldeado de tipos, uso de mecanismos de *boxing* y *unboxing* con el consiguiente coste asociado, dificultad de comprensión del código y mantenimiento, etc.

La influencia en lenguajes previos como C++, también incluido dentro de la familia de lenguajes de .NET, con el concepto de las plantillas (*templates*), o la existencia de lenguajes como EIFFEL con soporte para .NET, que incluyen dicha característica, apuntaban a una futura inclusión. El hecho puntual de que a partir de la versión 1.5 de JAVA, lenguaje multiplataforma directamente competidor con .NET, se incluyera la genericidad como parte del mismo, influyó en que a partir de la versión 2.0 del *framework* se incluyera la genericidad.

Aunque las cláusulas *tal que (where clauses)* no han sido incluidas en JAVA, sí han sido parcialmente añadidas en la plataforma .NET como mecanismo de acotación pero sólo para determinar la signatura del constructor por defecto que deben contener los argumentos de tipo.

Soporte actual Como se apunta en [Golding, 2005], la genericidad en .NET no aporta nuevas funcionalidades, sino una mejora en la forma de resolver los problemas a través de la seguridad de tipos, directamente relacionada con la calidad, facilidad de uso y mantenimiento del código, propiedades muy relacionadas con el contexto de la refactorización de código.

Aunque se apunte a la generación de un código más complejo, la realidad es que el código aporta más información sobre los tipos utilizados, lo que facilita la comprensión del código.

En .NET los tipos genéricos son instanciados en tiempo de ejecución por el entorno de ejecución (CLR), de forma dinámica, mientras que las plantillas C++ lo son en tiempo de compilación. De hecho, la versión 2.0 del *framework* .NET introduce una serie de extensiones al CLR para añadir dicho soporte en ejecución.

Creados bajo demanda, los tipos construidos con iguales sustituciones de los parámetros de tipo, comparten el código generado. Esto se consigue a través de un código intermedio (*Intermediate Language o IL*) que contiene referencias a los parámetros de tipo.

Frente a la solución de C++, de generar código para cada instanciación genérica en compilación, .NET genera código para cada instanciación en ejecución, permitiendo compartir el código entre aquellos tipos construidos que comparten iguales sustituciones de los parámetros de tipo.

Terminología En .NET y C# se utiliza la siguiente terminología:

Genéricos	Tipos (clases, estructuras o delegados) o métodos que son definidos en función de parámetros de tipo formal.
Parámetros de tipo	Parámetro utilizado en la definición de genéricos. Un genérico puede tener varios parámetros de tipo y la lista de parámetros definirá la signatura. Pueden ser aplicados a clases, estructuras, interfaces, delegados o métodos, pero no se pueden aplicar a indexadores, propiedades o eventos.
Argumentos de tipo	Tipos concretos proporcionados en la instanciación del tipo abierto.
Tipo construido	Representa una instancia concreta de un tipo abierto.
Tipo abierto	El tipo no está completamente definido, sino abierto a la sustitución del parámetro de tipo. En C++ se suele utilizar el concepto de tipo parametrizable.
Tipos construidos cerrados vs. abiertos	Cuando todas las sustituciones a los parámetros de tipo son tipos concretos se denominan tipo construido cerrado. Si por el contrario alguna de las sustituciones depende de algún parámetro de tipo se denomina tipo construido abierto.
Métodos genéricos	Métodos en cuya signatura se declaran parámetros de tipo.
Aridad	Número de parámetros de tipo usados por un genérico.

Variantes de acotación Las restricciones sobre los parámetros de tipos, surgen también en C# dada la limitación de la genericidad sin restricciones [Meyer, 1997]. Se introduce el concepto de restricciones o acotaciones para cualificar los parámetros de tipo, permitiendo al compilador asegurar que las invocaciones a las distintas operaciones sobre el parámetro de tipo son correctas.

Al respecto se establecen las siguientes reglas básicas:

- se permite acotación múltiple por interfaces.
- se permite acotación por una clase, como máximo.
- se permite acotación por tipos abiertos o construidos, pero no parámetros de tipos directamente.
- se permite añadir restricciones de constructor indicando la signatura del constructor que el argumento real debe contener.

Intentando clasificar el conjunto de restricciones de C# con las distintas variantes de acotación señaladas en la Sec. Apartado 2.2.1 se puede decir que cumple la acotación por subtipado, acotación F y en menor medida, también una forma de acotación con cláusulas tal que *(where clauses)*), referente a las restricciones de constructor por signatura.

Subtipado en genericidad En cuanto a las relaciones de subtipado, al igual que en JAVA, entre los tipos construidos se aplica la regla de no varianza (ni covariante ni contravariante en los argumentos reales). De tal forma, que `A<Y>` no es un subtipo de `A<X>`, pero `B<X>` sí que es un subtipo de `A<X>` si `B` es subtipo de `A` e `Y` es subtipo de `X`.

Esta limitación de no varianza, es dependiente del lenguaje particular dentro de la familia de lenguajes de .NET. Mientras que C# y VB .NET utilizan esta variante, otros lenguajes pueden elegir modelos más complejos, como por ejemplo EIFFEL en .NET que permite el subtipado en genericidad con covarianza. El CLR no impone limitaciones al respecto, quedando a disposición de cada lenguaje migrado a .NET la responsabilidad de escoger una u otra solución en presencia de genericidad.

Otro ejemplo de la diversidad de soluciones en .NET, es el caso de J# que aunque permite la instanciación de clases genéricas en su código, no permite la definición de las mismas a través del propio lenguaje [Golding, 2005]. Esto permite la definición de genéricos en otros lenguajes como C# o VB .NET, pero no en J#.

En la última especificación de C# [Microsoft, 2010], se ha modificado ligeramente la regla de no varianza, permitiendo sólo en interfaces y delegados la utilización de un modificador **`out`** para indicar covarianza e **`in`** para indicar contravarianza, entre los tipos actuales.

Genericidad en EIFFEL

Evolución Histórica EIFFEL, desde sus primeras versiones [Meyer, 1988], incluye los conceptos de tipos paramétricos, genericidad, clases genéricas y genericidad restringida. Sin embargo no se proporciona soporte a métodos genéricos y acotación múltiple. En base a dichas carencias, han sido propuestos algunos cambios en su especificación. Se realiza una nueva revisión en [Meyer, 1992], y finalmente, un esfuerzo de estandarización del lenguaje [Ecma International, 2006] donde se incluye la acotación múltiple a los parámetros genéricos y las restricciones en la creación de objetos, de forma similar a lo realizado en C#, aunque diferenciándose al poder declarar una lista de procedimientos de creación.

Además, desde un principio, introduce el concepto de **conformidad** entre tipos, que varía de las definiciones previas de subtipado. Toda clase conforma con sus ancestros respetando las reglas de herencia, donde se exige redefiniciones covariantes en atributos, argumentos y resultados de funciones, al contrario de la contravarianza en argumentos que exige el subtipado.

Soporte actual Introduce reglas adicionales de conformidad respecto a los parámetros de tipo como el uso de la palabra clave `frozen` y el símbolo `?` decorando las declaraciones de parámetros formales. El primero indica que para que se produzca conformidad se requiere igualdad de tipos en los argumentos reales, de forma similar a las reglas de subtipado en genericidad en C# y JAVA. El segundo, el símbolo ?, indica que la clase que se da como argumento contiene atributos que se auto-inicializan en su primer uso (deben permitir el acceso el método `default_create` definido en el ancestro `ANY` para su invocación).

Terminología En EIFFEL se utiliza la siguiente terminología [Meyer, 1992]:

Clases genéricas	Clases que contienen parámetros de tipo, que al ser instanciados (genéricamente derivados[4]) generan tipos paramétricos. Cualquier clase puede tener parámetros de tipo.
Parámetros genéricos formales	Parámetro utilizado en la definición de una clase genérica. Un tipo genérico puede tener varios parámetros de tipo y la lista de parámetros definirá la signatura del tipo. Estos parámetros se declaran junto con el código correspondiente a la declaración de la clase.
Tipos parametrizados	Tipos generados de la instanciación (genéricamente derivados) de una clase genérica.
Parámetro genérico real	Tipos proporcionados en la instanciación (derivación) de la clase genérica que pueden ser concretos o no.

Variantes de acotación Las restricciones sobre los parámetros de tipos se incluyeron en EIFFEL desde un principio, dadas las limitaciones de la genericidad sin restricciones [Meyer, 1997]. Se introdujo el concepto de restricciones o acotaciones: permitiendo cualificar los parámetros de tipo, permitiendo al compilador asegurar que las invocaciones a las distintas operaciones sobre el parámetro de tipo son correctas. Incluyendo adicionalmente una lista de procedimientos de creación para permitir que se pudiera contar con instrucciones de creación en el cuerpo de una clase genérica.

Intentando clasificar el conjunto de restricciones de EIFFEL con las distintas variantes de acotación señaladas en la Apartado 2.2.1 se puede decir que cumple la acotación por subtipado (en EIFFEL realmente se habla de conformidad al existir relación de herencia con el tipo acotación y covarianza entre los parámetros actuales), acotación F, y en menor medida, acotación con cláusulas tal que *(where clauses)*, referente a las restricciones de constructor por signatura.

Subtipado en genericidad Desde el punto de vista de la genericidad, una clase genérica `B` conforma con una clase genérica `A`, si `B` conforma con `A` y además todos sus parámetros reales de `B` instanciada (derivada) conforman con los parámetros reales de A instanciada. En otros lenguajes se denomina a esta relación "tipos genéricos covariantes". Como se ha comentado previamente, el uso del modificador `frozen` fuerza a la no varianza entre los parámetros actuales (impide la covarianza).

Genericidad en JAVA

Evolución histórica A lo largo del tiempo, ha habido varias propuestas para la inclusión de genericidad en JAVA. Generic Java (GJ) [Bracha et al., 1998a, Bracha et al., 1998b, Bracha et al., 1998c] se adoptó finalmente como punto inicial para la inclusión en JAVA de la genericidad. Desde su primera propuesta fueron incluidos los conceptos de clases genéricas, parámetros de tipo y acotaciones (acotación por subtipado y acotación F).

En esta línea de trabajo, otra propuesta como Poly J [Liskov et al., 1995, Liskov et al., 1998] incluía los mismos conceptos aunque con diferente sistema de acotación utilizando cláusulas *tal que (where clauses)*). Para cada parámetro de tipo, se podían enumerar las signaturas de los métodos que deben contener las sustituciones reales. Sin embargo dicha propuesta no fue finalmente incluida en la version final.

GJ se adoptó con algunos cambios finales para mantener los requisitos de compatibilidad hacia atrás, eficiencia y robustez. Se incluye en las implementaciones de los entornos de desarrollo y ejecución de Sun Microsystems (liberadas en Otoño del 2004) a partir de la versión JAVA 1.5 (también conocida como "Tiger" o JAVA 5.0), con ligeras modificaciones a la propuesta inicial.

Entre las modificaciones incluidas finalmente, se debe señalar que en JAVA se aligera la restricción de no varianza en la relación de subtipado entre tipos paramétricos permitiendo utilizar el tipo ? con la semántica de "cualquier tipo" permitido. Por ejemplo, para un tipo definido como `List<?>` son subtipos válidos `List<String>` y `List<Point>` aunque no existe ninguna relación de herencia entre `String` y `Point`. Como limitación de su uso, las colecciones son de sólo lectura y se restringen las propiedades a utilizar, en este ejemplo concreto a las propiedades de la clase `Object`.

En la última versión de JAVA (versión 7) se ha incluido una pequeña modificación, incluyendo el operador diamante <> para la inferencia de tipos en las instanciaciones de tipos derivados de clases genéricas. De esta forma no es necesario repetir los tipos en la instanciación, sino que se infieren a partir de la declaración estática de tipo de la variable. Por ejemplo, tomando la instanciación:

```
Map<String, List<String>> map = new HashMap<String, List<String>>();
```

donde `HashMap` es descendiente de `Map`, en la versión JAVA 7 se puede reescribir como:

```
Map<String, List<String>> map = new HashMap<>();
```

donde el operador diamante (<>) cumple el papel de "azúcar sintáctico" simplificando la instanciación.

Soporte actual La especificación completa, incluyendo la descripción de la genericidad como parte del lenguaje, pero sin incluir al operador diamante, se encuentra en su tercera edición (Java Language Specification - JLS3) disponible en [Gosling et al., 2005].

Desde un punto de vista del compilador de JAVA, se opta por eliminar toda información sobre los parámetros de tipo y argumentos de tipo del fichero binario generado. Mediante un proceso de borrado de tipo (*type erasure*) se genera el correspondiente tipo crudo (*raw type*). Por ejemplo, a partir de un tipo paramétrico `List<String>` se genera el tipo `List`.

Todas las instanciaciones de la misma clase genérica (tipos paramétricos) comparten el mismo tipo en tiempo de ejecución (el tipo crudo). Esa pérdida de información de tipos que se produce, limita y restringe ciertas acciones, como el uso de genericidad con *arrays* y que la idea de tipo reificados, disponible en otros lenguajes, no se pueda llevar a cabo.

Terminología En JAVA se utiliza la siguiente terminología [Gosling et al., 2005]:

Tipos genéricos	Tipo que contiene parámetros de tipo, que al ser instanciado genera tipos paramétricos. Cualquier clase o interfaz puede ser un tipo genérico, es decir, puede tener parámetros de tipo, excepto los tipos enumerados, las clases internas anónimas y las clases que implementan excepciones.
Parámetros de tipo formal	Parámetro utilizado en la definición del tipo genérico. Un tipo genérico puede tener varios parámetros de tipo y la lista de parámetros definirá la signatura del tipo. Pueden ser utilizados en las declaraciones de clases y métodos.
Tipos parametrizados	Tipos generados de la instanciación de un tipo genérico.
Tipos parametrizados con comodines	Tipos generados de la instanciación de un tipo genérico utilizando el tipo desconocido o tipo comodín (representado en el código con el carácter ?).
Tipos parametrizados concretos	Representa una instancia de un tipo genérico donde las sustituciones a sus parámetros de tipos son tipos concretos y no tipos comodín. Se pueden utilizar para declaración de tipos excepto para *arrays*, excepciones, literales de clases y con la instrucción `instanceof`.
Tipo crudo	Tipo genérico instanciado sin dar sustitución a los parámetros de tipo. Se mantiene por motivos de compatibilidad hacia versiones previas. Se produce cuando se utiliza una clase o interfaz genérica, pero se utiliza sin tener en cuenta su parámetros de tipo. Por ejemplo, la clase genérica `java.util.HashMap<K,V>` puede ser instanciada de la forma **`new`** `HashMap();` sin dar sustituciones a los parámetros `K` ni `V`.
Argumentos reales de tipo	Tipos concretos proporcionados en la instanciación del tipo genérico. No se permiten tipos primitivos.
Métodos genéricos	Métodos en cuya signatura se declaran parámetros de tipo.
Acotación superior	Se acotan las sustituciones de manera que tendrán que ser subtipos del tipo cota superior.
Acotación inferior	Se acotan las sustituciones de manera que tendrán que ser supertipos del tipo cota inferior.

Se incluye alguna norma adicional: los parámetros de tipo no pueden ser utilizados en ningún contexto estático de las clases, los tipos paramétricos no son covariantes y no se puede utilizar genericidad en la construcción de excepciones.

Variantes de acotación Se introduce el concepto de acotaciones o restricciones: se cualifican los parámetros de tipo, de tal forma que el compilador puede comprobar la corrección

de las invocaciones a las distintas operaciones, sobre expresiones cuyo tipo es el parámetro de tipo.

Al respecto se establecen reglas básicas:

- se permite acotación múltiple por interfaces.
- se permite acotación por una única clase.
- se permite acotación por tipos parametrizados concretos o no, y tipos enumerados, pero no por tipos primitivos y *array*s.

Intentando clasificar el conjunto de restricciones de JAVA, con las distintas variantes de acotación señaladas en la Apartado 2.2.1, se puede decir que cumple la acotación por subtipado y acotación F.

Subtipado en genericidad Las reglas de subtipado en genericidad no comprueban los argumentos reales, sino que establece la regla de no varianza en los argumentos (ni covariante ni contravariante en los argumentos reales). De tal forma que, siendo `B` subtipo de `A` e `Y` subtipo de `X` , `A<Y>` no es un subtipo de `A<X>`, pero `B<X>` sí que es un subtipo de `A<X>`.

Estudio comparativo

Aunque cada uno de los lenguajes descritos tiene sus particularidades, la mayoría comparten un núcleo común de propiedades que permiten una definición reutilizable de las refactorizaciones aplicadas a la programación genérica. La Tabla 2.6 analiza la situación actual de la inclusión de propiedades en cuanto a genericidad, en las versiones de C++ (o C++11 [Becker, 2011]), C# (en su version 4.0 [Microsoft, 2010]), EIFFEL (con su estándar ECMA-367 [Ecma International, 2006] y JAVA [Gosling et al., 2005] (en la última version, Java 7, se han incluido algunas pequeñas modificaciones en la sintaxis –e.g. operador diamante en genericidad– , pero no existe una especificación actual que la incluya).

Tabla 2.6: Propiedades generales sobre genericidad en los LPOO principales

	Lenguajes			
Propiedad genérica	C++	C#	EIFFEL	JAVA
Clase	+	+	+	+
Método	+	+	-	+
Acotación por subtipado/conformidad	-	+	+	+
Acotación F	-	+	+	+
Acotación múltiple	-	+	+	+
Acotación por signatura (tal que)	-	*	*	-

En la Tabla 2.6, un asterisco (*) indica que la propiedad ha sido implementada parcialmente, en particular con la acotación por constructores de los parámetros formales, pero no incluyen un soporte completo a la acotación por cláusulas tal que. El signo más (+) indica que está disponible, un signo menos (-) indica que no está disponible (aunque puede estar en estudio su inclusión en próximas revisiones como la acotación en C++).

Como se puede concluir, las semejanzas entre los lenguajes de la corriente principal en cuanto a características relativas a la programación genérica, ofrecen un punto de arranque a la hora de proponer y poner en marcha una solución a la ejecución de refactorizaciones con cierta independencia del lenguaje.

2.2.4. Relevancia de la programación genérica: refactorización en genericidad

El auge que se puede prever de la programación genérica, apunta a que también debe ser un punto de interés a tratar en la refactorización de código. Si bien hasta ahora, el hecho de no estar presente en lenguajes de la corriente principal puede haber frenado esta posibilidad. La evolución en los últimos años indica un campo de interés y un cierto vacío en la investigación al respecto, aunque esa tendencia está cambiando como se puede observar de la evolución de trabajos en esta línea.

En [Siff and Reps, 1996], los autores se centran en traducir funciones C a plantillas de funciones en C++, usando inferencia de tipos para detectar polimorfismo latente. Por otro lado, en [Duggan, 1999] se aporta un algoritmo para modificar clases hacia la variante genérica de Java denominada PolyJ [Myers et al., 1997, Liskov et al., 1998], diferente de la propuesta final incorporada al lenguaje Java.

En [Crespo, 2000], se define una refactorización **parametrizar**, que a partir de una clase y una entidad guía, realiza el proceso de añadir un parámetro formal en la clase a partir de la información de entrada. La refactorización se define sobre una notación minimal para LOO con el fin de definir la refactorización de forma independiente del lenguaje. Posteriormente se valida su aplicación sobre el lenguaje Eiffel. Este trabajo presenta uno de los primeros intentos por lograr cierta independencia del lenguaje en refactorización.

En [Géraud and Duret-Lutz, 2000], se utiliza la genericidad sobre patrones de diseño, para mejorar el diseño. El trabajo se enfoca desde el punto de vista de mejora de eficiencia, en particular sobre bibliotecas de cálculo científico. La mejora de eficiencia no es un objetivo de la refactorización. Sin embargo este trabajo sí que presenta los posibles beneficios de cambiar el software hacia soluciones más genéricas. El autor centra la mejora en sustituir el uso del polimorfismo de inclusión y la ligadura dinámica, por el uso de tipos paramétricos, con la generación de las instancias genéricas en compilación. Este último punto también ha sido discutido por otros autores (ver [Meyer, 1997]) y se trata de una aproximación hacia la evolución de software usando genericidad. El trabajo refleja tres variaciones de patrones de diseño usando genericidad: **Generic Iterator**, **Generic Template Method** y **Generic Decorator**.

En [Marticorena et al., 2003], se amplió el trabajo previo de [Crespo, 2000], definiendo un catálogo, de manera semiformal, de refactorizaciones sobre clases genéricas. En particular, desde el punto de vista de la especialización de clases. Entendiendo por especialización la eliminación o cambio a contextos más específicos (generalmente a descendientes en una jerarquía de herencia), de elementos estructurales que dificultan la comprensión y aumenten la complejidad del diseño, sin alterar el comportamiento inicial de las clases en el contexto dado.

En [von Dincklage and Diwan, 2004], lo autores convierten clases JAVA no genéricas a genéricas (parametrización similar a [Crespo, 2000]), y actualiza el uso de dichas clases (instanciación genérica). Basado en heurísticas, consigue un cierto éxito en la parametrización. Resulta necesario eliminar manualmente elementos no tratados por la herramienta, y se asumen algunas suposiciones que pueden afectar al comportamiento posterior.

Continuando la línea de trabajos anteriores, en [Donovan et al., 2004], se diseña e implementa una refactorización para migrar una aplicación para que utilice la versión genérica de la biblioteca básica de JAVA (`java.util`). Se infieren los tipos a partir de colecciones estándar en un conjunto de productos JAVA. La medida de su éxito se establece en el número de moldeados (*cast*) que pueden ser eliminados.

En esta misma línea, en [Tip et al., 2003, Tip et al., 2004, Fuhrer et al., 2005], se estudia el problema de la migración de código JAVA de su versión 1.4 y anteriores a la versión 1.5. A partir de su versión 1.5, las bibliotecas de clases de JAVA incluían el uso de la genericidad. Sin embargo el código legado, utilizaba el tipo en lo que equivale al concepto de "tipo crudo" (*raw type*) [Bracha et al., 2001]. Como solución se propone refactorizar el código cambiando el uso de los tipos crudos por sus correspondientes versiones genéricas. El trabajo se integra en ECLIPSE. De los resultados obtenidos se observa la eliminación de los *downcast* y avisos del compilador en cuanto al peligro en corrección de tipos, mejorando los resultados previos obtenidos en [Donovan et al., 2004].

Estos autores en [Kiezun et al., 2007] abordan el problema de parametrizar clases ya existentes, comparando sus resultados con las actuales implementaciones de bibliotecas genéricas. Este trabajo se encuentra en la línea de los trabajos de [Crespo, 2000, von Dincklage and Diwan, 2004], y en particular centrado en JAVA e implementado sobre ECLIPSE.

La inclusión de la genericidad afecta también a refactorizaciones no directamente ligadas a dicha característica. En [Marticorena et al., 2010a] se realiza un estudio de la refactorización **EXTRACT METHOD** aplicada en distintos entornos de desarrollo sobre código con genericidad, en concreto centrándose en entornos JAVA. Como resultado de dicho trabajo, se concluye que no en todos los entornos se han tenido en cuenta aspectos relevantes a la genericidad en la definición y construcción de las refactorizaciones. En muchos casos los cambios realizados no generaban código correcto después de la transformación, al no tener en consideración la inclusión de código con genericidad.

Como se puede observar, existe un amplio campo abierto ante la posibilidad de refactorizar código teniendo en cuenta las características de la programación genérica y el uso de clases y métodos genéricos. Aunque el trabajo está iniciado, quedan muchos aspectos y puntos a considerar. En su mayoría los trabajos se han orientado a un único lenguaje, y se ha

adolecido de falta de integración en herramientas dedicadas de refactorización o entornos de desarrollo. El presente trabajo parte de estas premisas, para llegar a dar una solución generalizable en la refactorización aplicada a los lenguajes de la familia de LPOO estáticamente tipados.

CAPÍTULO 3

INDEPENDENCIA DEL LENGUAJE EN REFACTORIZACIÓN

Al abordar el problema de la aplicación de refactorizaciones, es necesario acotar el nivel de abstracción al que se va a trabajar. Esto determina sobre qué productos concretos se van a llevar a cabo las transformaciones. En particular, en el presente trabajo, las refactorizaciones se establecen al nivel de implementación, por lo tanto transformando el código fuente. Más aún, se persigue el objetivo de refactorizar con cierta independencia del lenguaje de programación utilizado en la codificación, partiendo de la base establecida en [Crespo, 2000].

Bajo estas condiciones previas, otras cuestiones adicionales deben ser tomadas en cuenta como son:

- el paradigma del lenguaje de programación.
- las características del lenguaje en cuanto a tipado (estático vs. dinámico).
- las herramientas implicadas en el proceso de refactorización.

Definir e implementar una refactorización de manera completamente independiente al lenguaje es imposible, como ya se estableció en otros trabajos [Van Gorp et al., 2003a, Mendoça et al., 2004, Nierstrasz et al., 2005]. Esto se debe a que la refactorización es aplicada en última instancia sobre código real, con sus características particulares asociadas. Estas características son en gran medida dependientes del lenguaje, puesto que en muchos casos las propiedades inherentes al lenguaje objetivo influyen tanto en el cumplimiento de las condiciones previas, como en las acciones necesarias a aplicar para poder efectuar la refactorización sobre el código fuente.

La independencia del lenguaje sería viable si los lenguajes fuesen equivalentes o isomorfos entre sí, con características sintácticas y semánticas equivalentes. En un escenario ideal esto permitiría la traducción entre lenguajes de programación de una manera simple. Resultados de este tipo se obtienen, por ejemplo, en la plataforma .NET a través de herramientas que permiten la traducción en ambos sentidos entre los lenguajes C# y VB .NET. Este es un caso particular, puesto que ambos lenguajes nacen a partir de un punto común como es la plataforma .NET y con el mismo paradigma de programación.

Sin embargo, esto es difícilmente viable en la mayoría de los casos, puesto que las diferencias existentes entre los lenguajes son en algunos casos insalvables. Esto se agudiza ante diferentes paradigmas (*e.g.* programación orientada a objetos vs. programación lógica). Incluso dentro del mismo paradigma se encuentran diferencias sustanciales en su sistema de tipos: declaración explícita de tipos o no, tipado dinámico o estático, lenguajes fuertemente tipados o relajados, etc. Considerando el mismo paradigma, con declaración de tipos estática, el sistema de tipos subyacente puede ser diferente (covarianza vs. contravarianza en argumentos o tipos de retorno en signaturas de métodos).

Acotando más el ámbito de trabajo, centrando el esfuerzo en lenguajes orientados a objetos con declaración estática de tipos y con distintas variantes de sistemas de tipos, las diferencias siguen impidiendo un enfoque totalmente independiente del lenguaje de programación. Aún con ciertos puntos en común, es habitual que cada lenguaje introduzca elementos propios, características particulares, etc. no definidos en el resto de lenguajes, y que dificultan un enfoque de total independencia del lenguaje en su refactorización.

Sin embargo, aunque las diferencias existen, se pueden encontrar algunos puntos en común. Una vez acotado el conjunto de lenguajes objetivo (lenguajes orientados a objeto estáticamente tipados) existen conceptos básicos comunes a todos como: espacios de nombres, tipos, clases, propiedades (básicamente atributos y métodos), modularidad, encapsulación, herencia, genericidad, contratos, excepciones, etc. Estos conceptos son bien conocidos y pueden ser expresados de forma general para un conjunto amplio de lenguajes, para luego ser representados de forma particular en cualquiera de ellos (*e.g.* C++, Java, Eiffel, C#, VB .NET, etc.).

Un estudio sobre las similitudes y diferencias de los conceptos de tipos, clases, subtipo, herencia y genericidad entre un conjunto amplio de lenguajes está presente en [Crespo, 2000]. Otra obras como [Joyner, 1999, Eliens, 2000], han realizado este tipo de comparativas en un contexto de aplicación práctica acotado a C++, Java y Eiffel. Comparativas más recientes también están disponibles en [Meyer, 2009], incluyendo C# entre los lenguajes objeto de su estudio. En particular, estudios sobre genericidad y o su futura inclusión están presentes en trabajos como [Garcia et al., 2003]. Finalmente, una revisión de la situación actual en LPOO en cuanto a la programación genérica puede consultarse en el Capítulo 2, en su Apartado 2.2.3.

Por lo tanto, dado que existen diferencias, pero también puntos comunes entre los diferentes lenguajes de una misma familia, cabe plantearse la posibilidad de brindar una solución a la definición, construcción y ejecución de refactorizaciones con el mayor grado posible de reutilización.

Como solución de representación para el análisis de las refactorizaciones, y en algunos casos para su aplicación, se han planteado soluciones basadas en lenguajes y metamodelos con cierto carácter minimal. Soluciones como FAMIX [Demeyer et al., 1999], MOON [Crespo, 2000], GrammyUML [Van Gorp et al., 2003a] o RefaX [Mendoça et al., 2004] intentan aprovechar esos puntos en común para obtener un cierto grado de reutilización en los esfuerzos realizados.

En base a esta idea se definen las refactorizaciones de forma general, empleando conceptos generales, para posteriormente particularizar su solución al lenguaje concreto. Este tipo de cuestiones también se han afrontado con éxito en la implementación de los patrones de diseño en diferentes lenguajes [Jézéquel et al., 1999, Metsker, 2004, Shalloway and Trott, 2004, Bishop, 2008]. El patrón se esboza de forma general, pero la forma de llevarse a cabo puede ser diferente en cada lenguaje (denominado como *idioms* en la literatura de patrones), incluso pueden existir diferentes variantes dentro del mismo lenguaje.

De lo anterior se concluye que es necesario mantener esa doble visión de dependencia e independencia del lenguaje en las refactorizaciones, permitiendo además una aplicación práctica y real. En la Sec. 3.1, se establece el marco de referencia que se utilizará a lo largo del presente trabajo para abordar el problema desde la independencia del lenguaje, para en la Sec. 3.2, a partir de dicho marco, exponer la arquitectura diseñada, con el fin de dar una solución racional y práctica al problema de la integración de los elementos dependientes del lenguaje objetivo.

3.1. MOON como marco de referencia

En [Crespo, 2000] se presentó una notación minimal para lenguajes orientados a objetos denominada MOON. Como resultado de dicho trabajo *"se define un marco de referencia específico para desarrollar refactorizaciones con el objetivo de transformar elementos de implementación, escritos en un lenguaje de programación orientado a objetos (LPOO), estáticamente tipado y con genericidad"*.

MOON se definió con ciertas propiedades generales que permiten el soporte de muchas características comunes a los LPOO estáticamente tipados como:

- declaración de tipos.
- clases.
- creación de objetos.
- envío de mensajes.
- constantes manifiestas.
- asociaciones de objetos a entidades:
    - mediante instrucciones de creación de objetos.

  - mediante asignación de expresiones a variables.
  - mediante paso de expresiones como argumentos reales en el envío de mensajes a los objetos.

- herencia simple y múltiple.
- genericidad.

Por otra parte, en relación a esta última propiedad, se definieron tres variantes de sistema de tipos en relación a las formas de acotación de los parámetros formales:

- subtipado (incluyendo acotación F): consiste en indicar un tipo asociado al parámetro genérico formal. Todo parámetro real debe ser un subtipo del tipo utilizado como acotación. La acotación F es una extensión que permite acotar con tipos recursivos.
- cláusulas *tal que* (*where clauses*): se asocia al parámetro formal una cláusula con las signaturas de las propiedades que debe tener todo tipo que se quiera utilizar como parámetro real.
- conformidad: similar a la acotación por subtipado exigiendo que todo parámetro real conforme con el tipo utilizado como acotación. El concepto de conformidad se toma de EIFFEL [Meyer, 1997], donde toda clase conforma con sus ancestros respetando las reglas de herencia donde se exige covarianza en la redefinición de atributos, argumentos y resultados de funciones. Además se considera que una clase genérica B instanciada conforma con una clase genérica A instanciada si la clase B conforma con A y los parámetros reales de B conforman con los parámetros reales de A.

A partir de la gramática de MOON, se estableció una refactorización **PARAMETERIZE**, que permite parametrizar una clase a partir de una entidad guía. La solución se validó sobre el lenguaje EIFFEL y se implementó un prototipo para demostrar la adecuación de la solución.

El presente trabajo continúa a partir de la base establecida con MOON, abordando algunas de las cuestiones abiertas en [Crespo, 2000]:

- definición de otras refactorizaciones de acuerdo al marco de referencia de MOON.
- continuar con el estudio, implementación y mejora de herramientas de soporte a la refactorización con cierta independencia del lenguaje.

En relación al segundo punto abierto anteriormente, la línea de trabajo iniciada en [López and Crespo, 2003] optó por definir una solución basada en *frameworks* [Fayad et al., 1999] como solución práctica para MOON. Este punto será desarrollado en el Capítulo 6.

Por lo tanto, partiendo de la gramática inicial del lenguaje modelo, se trabajó en el diseño de un metamodelo que recoja todos los elementos y relaciones definidos en la gramática, así como en su ampliación o modificación en aquellos puntos en los que la notación minimal se

ha considerado que debía ser extendida, según lo presentado en [Marticorena and Crespo, 2003] y observaciones recogidas a lo largo del desarrollo del presente trabajo.

En la siguientes secciones, se presenta el modelado de las características fundamentales (a través de abstracciones) definidas en el lenguaje modelo MOON, dando una breve explicación de su rol en el modelado de lenguajes de programación.

3.1.1. De la gramática MOON al metamodelo MOON

La gramática de MOON está descrita con una variante de LDL [ISE, 1992] (ver Apéndice A). Si bien no sigue la especificación estricta de *Extended Backus-Naur Form* (EBNF) [ISO, 1996], se puede tratar de manera equivalente a otras gramáticas expresadas en EBNF. Esto es de particular interés cuando se aborda el problema de la transformación de una gramática a un metamodelo, como es nuestro caso. Dicho problema ha sido tratado en la literatura [Alanen et al., 2003, Wimmer and Kramler, 2006, Kunert, 2008, Izquierdo and Molina, 2009] intentando dar una solución a la transición entre el mundo software de las gramáticas (*grammarware*) y de los metamodelos (*modelware*).

Sin entrar en un estudio tan profundo como la generación de *parsers* para gramáticas y lenguajes concretos, afrontada en la bibliografía mencionada, se ha aplicado el algoritmo *ad-hoc* propuesto en [Kunert, 2008] así como las soluciones de reducción y simplificación (similares a las ideas de condensación y personalización de los metamodelos de [Wimmer and Kramler, 2006]). Con esto se logra, a partir de la gramática MOON, generar el metamodelo equivalente.

Como [Kunert, 2008] establece, el objetivo es obtener un *"metamodelo de calidad"*, ligando la calidad al objetivo de uso del metamodelo. MOON partía como lenguaje minimal para el análisis de refactorizaciones para una familia de lenguajes de programación orientados a objetos, y ese objetivo debe seguir siendo inherente al metamodelo obtenido.

El algoritmo parte de unas reglas de transformación básicas como son:

- generar una clase para cada elemento de la gramática (terminal y no terminal).
- introducir clases adicionales para secuencias, alternativas, opciones y repeticiones.
- conectar las clases generadas de acuerdo a las reglas de la gramática (usando agregaciones, asociaciones o generalizaciones).

De una primera iteración, aplicando estas reglas básicas, se obtiene una primera versión del metamodelo pero, en palabras del autor, la calidad del mismo depende del grado de satisfacción con el propósito del mismo. Dada la naturaleza minimal de MOON, el objetivo del metamodelo es representar la semántica y no la sintaxis (más aún cuando hablamos de un lenguaje no concreto) con un número mínimo de clases.

Como primera simplificación, se parte de una sintaxis abstracta eliminando de MOON todos los terminales ligados a su sintaxis concreta. El objetivo del lenguaje no es disponer de una implementación real para ser usado en programación. Su objetivo es el análisis y

definición de refactorizaciones. Siguiendo el trabajo de [Kunert, 2008], también las construcciones resultado de las clases auxiliares generadas por opciones, repeticiones, secuencias y alternativas, generadas por el algoritmo, se simplifican eliminándose o representándose en las multiplicidades de las asociaciones.

Igualmente, propone la eliminación de identificadores, reemplazando los mismos por referencias – asociaciones – al objeto referenciado. En nuestro caso esto implica que los elementos terminales de la gramática como `METHOD_ID`, `FORMAL_GEN_ID`, etc. serán reemplazados por una asociación correspondiente al elemento correspondiente a `METH_DEC`, `FORMAL_PAR`, etc. donde se declararon.

Finalmente el autor propone el uso de conceptos abstractos como por ejemplo el uso de espacios de nombres (*namespace*) en lenguajes de programación para representar conceptos presentes en distintos lenguajes. Esta última premisa ya había sido tenida en cuenta y fruto de estos razonamientos se observan las modificaciones previas realizadas a la gramática en [Marticorena and Crespo, 2003].

Partiendo de dichos trabajos, de los resultados ya obtenidos previamente del diseño del *framework* presentado en [López and Crespo, 2003], de la revisión de la gramática realizada en [Marticorena and Crespo, 2003] y de la validación aplicando [Kunert, 2008], se presenta el metamodelo MOON que servirá como base fundamental en este trabajo. Por motivos de claridad, su exposición se divide en distintas secciones, agrupando los elementos en base a un criterio de semántica común.

3.1.2. Tipos y clases

Se modela el concepto fundamental de tipo en LPOO estáticamente tipados como `Type`. El identificador del tipo que una clase implementa (`ClassDef`), depende del propio nombre simple de la clase que lo genera. Dada la presencia de genericidad en MOON, el concepto de tipo se especializa en parámetros formales (`FormalPar`) y tipos implementados por las propias definiciones de las clases (`ClassType`) (ver Fig. 3.1).

Cuando la definición de una clase no es genérica la asociación entre la clase (`ClassDef`) y el tipo implementado por la clase (`ClassType`) es uno a uno, es decir se considera a una clase como una construcción lingüística en un LPOO que se usa para implementar los tipos. Por tanto el nombre simple y único del tipo vendrá dado por el nombre de la clase, existiendo coincidencia entre ambos.

Cuando la definición de una clase es genérica, se puede decir que es una clase determinante de un conjunto potencialmente infinito de tipos. La clase genérica define un conjunto de parámetros formales de tipo. Cada una de las instanciaciones genéricas realizadas (aportando una sustitución para sus parámetros formales) se corresponde con distintos tipos (`ClassType`).

Las clases suelen ser la unidad modular que aglutina una serie de propiedades o características como un todo. Esto se modela mediante la inclusión de propiedades (`Property`), bien como atributos (`Attribute`) o métodos (`Method`), como elementos básicos que constituyen una clase. La correspondencia de las propiedades con los atributos se deduce de la gramática de MOON. Aplicando la regla de sustitución de un identificador por la declara-

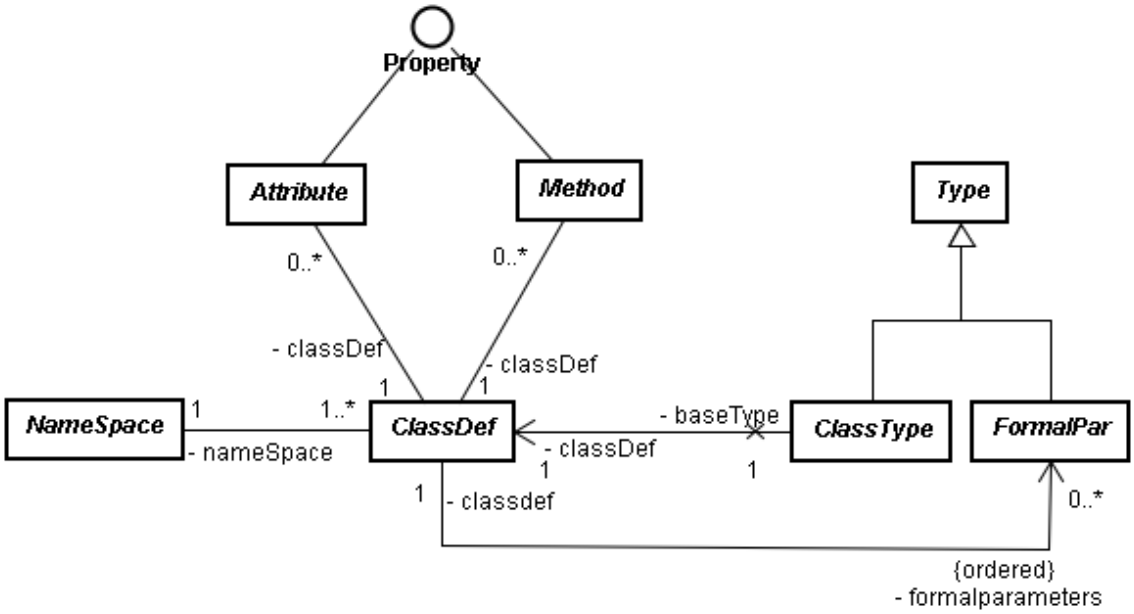

Figura 3.1: Tipos y clases

ción de su elemento declarado correspondiente [Kunert, 2008], se sustituye el identificador de variable por la declaración de atributo y el identificador de método por la declaración del mismo.

En los entornos reales de programación se proporciona algún mecanismo que sirve para identificar de forma unívoca una clase añadiendo información referente al contexto de la clase. Este es el caso de `package` en JAVA, `cluster` en EIFFEL, `namespace` en .NET, etc. Dicho concepto se recoge en el metamodelo con el concepto de `NameSpace`, que contendrá un conjunto de clases, permitiendo recoger los contextos de nombres y evitando sus colisiones. Permite representar y obtener el nombre completamente cualificado de un tipo, en contraposición con su nombre simple.

3.1.3. Entidades

Una entidad es un nombre (un identificador) en el texto de una clase que denota un objeto (valor o referencia) en tiempo de ejecución con un tipo asociado. En la Fig. 3.2 se muestra la clasificación de las entidades, añadiendo un nivel donde se consideran las entidades predefinidas (`PredefinedEntity`), entidades internas (`InternalEntity`) y entidades de signatura (`SignatureEntity`).

Las entidades básicas consideradas en MOON, a partir de su descripción de la sintaxis abstracta y dentro de la jerarquía de herencia mencionada previamente, son: entidades pre-

definidas como la auto-referencia (`Self`), la referencia paterna (`Super`) o la entidad ligada al resultado de una función (`Result`), variables locales a un método (`Local`), argumentos formales de un método (`FormalArgument`) y atributos (`Attribute`).

Partiendo de la gramática de MOON se observa que tanto `Attribute`, `Local` y `FormalArgument` tienen asociaciones con un elemento de tipo *declaración de variable* con tipo explícito. Dada la asociación del identificador de variable en `Entity` con dichas declaraciones de variable, se deduce la relación de herencia mostrada en la Fig. 3.2.

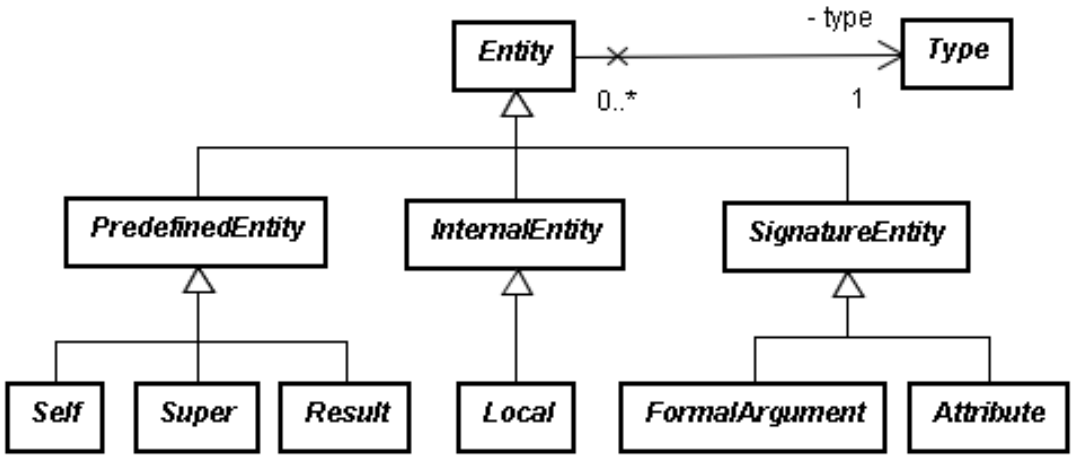

Figura 3.2: Entidades

El trabajo de [Crespo, 2000] se extendió en [Marticorena and Crespo, 2003] introduciendo modificaciones a la gramática de MOON para introducir el concepto de referencia paterna (`Super`), añadiéndose como entidad predeterminada, puesto que se recoge en la práctica totalidad de LPOO (*e.g.* **`super`** en JAVA, **`Precursor`** en EIFFEL, **`base`** en C#, **`MyBase`** en VB .NET, etc.)

Por otro lado se debe señalar el diferente ámbito de las entidades:

- Los atributos (`Attribute`) y entidades predefinidas (`Self` y `Super`) tienen un contexto de clase. El tipo de `Self` siempre será igual al tipo de la clase en la que se utiliza. El tipo de `Super` será siempre el tipo de la superclase o ancestro directo. En presencia de herencia múltiple se sugiere en [Marticorena and Crespo, 2003] el uso de una anotación explícita de tipo para resolver conflictos.

- Las entidades locales (`Local`), los argumentos formales (`FormalArgument`) y el resultado (`Result`), tienen un contexto de método. Las entidades de resultado siempre están ligadas a una función, nunca a una rutina.

3.1.4. Métodos

MOON soporta dos tipos de métodos (ver Fig. 3.3): los que tienen un valor de retorno, denominados *funciones* (`Function`), y los que no, denominados *rutinas* (`Routine`).

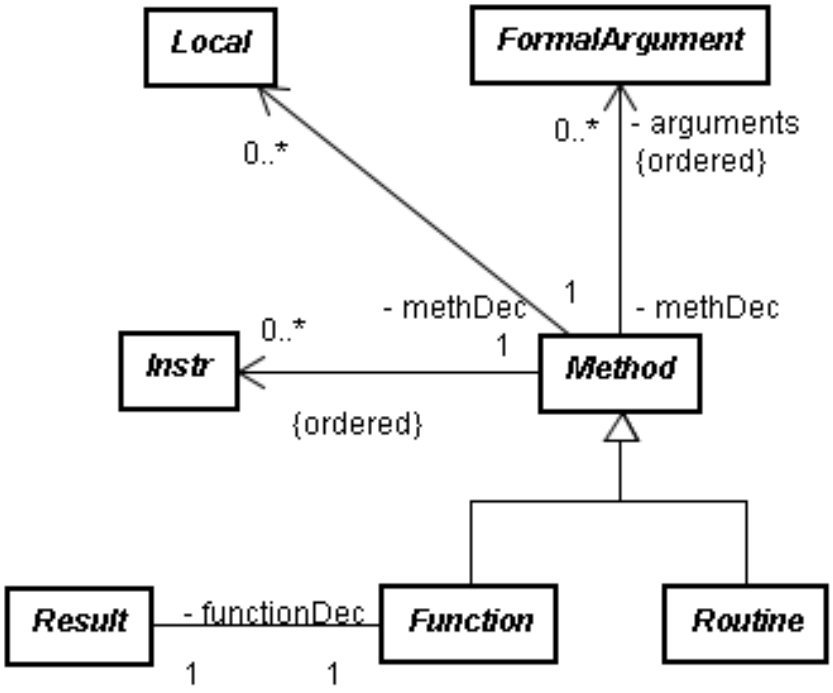

Figura 3.3: Métodos

La descripción de los métodos está determinada por su nombre y signatura. Las signatura se corresponde con la lista ordenada de tipos de sus argumentos formales (`FormalArgument`) y la entidad de retorno (ligada a un tipo) en el caso de tratarse de una función[1]. Cada argumento se corresponde con una entidad y su tipo. Aplicando las reglas de tipos correspondientes se determinará la compatibilidad entre signaturas de métodos.

La gramática de MOON separa claramente la declaración de métodos de su implementación, pero al aplicar las simplificaciones sugeridas en [Kunert, 2008], se ha optado por unificar en el metamodelo la declaración e implementación. De esta forma un método, no sólo tiene signatura, sino que se asocia también el cuerpo del mismo si lo hubiera.

La implementación de un método va a estar compuesta por un conjunto de variables locales (`Local`) y un conjunto de instrucciones ordenadas (`Instr`) que conforman lo que

[1]El concepto de signatura de método varía entre los diferentes lenguajes siendo tarea de la extensión concreta a MOON determinar la semántica particular del concepto de signatura.

se denomina como cuerpo del método (una descripción detallada de las instrucciones se encuentra en el Apartado 3.1.7).

3.1.5. Genericidad

MOON incluye el concepto de tipos paramétricos y clases genéricas, permitiendo que una clase tenga parámetros formales y genere un número ilimitado de tipos como resultado de la instanciación concreta de dichos parámetros por otros tipos (ver Fig. 3.4).

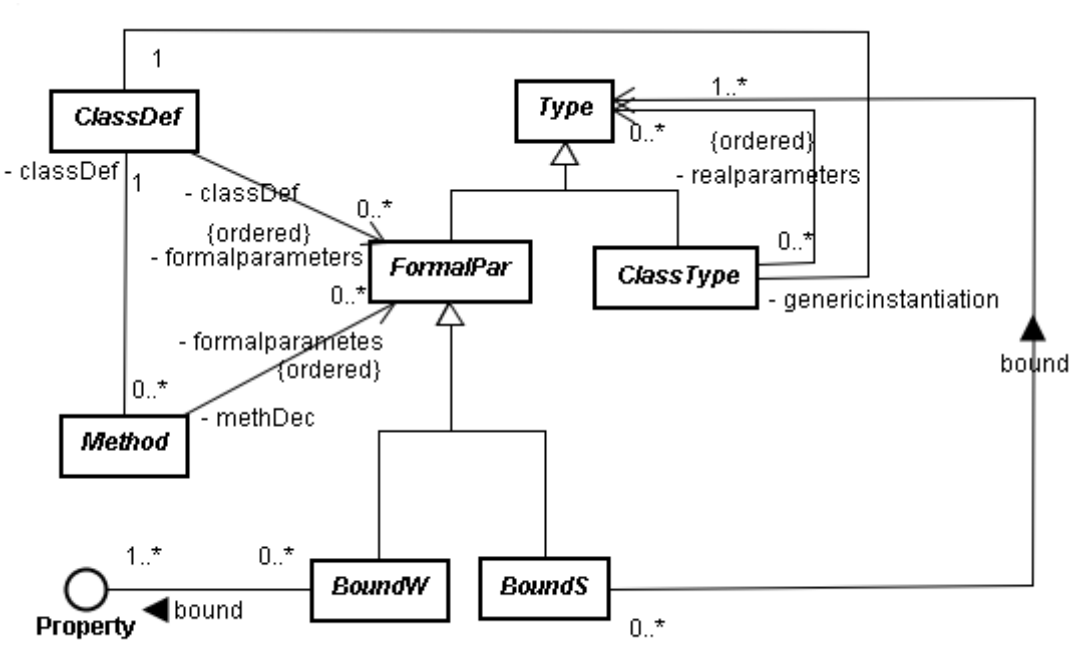

Figura 3.4: Genericidad

Cada una de las instanciaciones genéricas realizadas se corresponde con distintas instancias de la clase `ClassType`, de esta forma la definición de una clase genérica representa un conjunto de tipos.

Como variación al trabajo de [Crespo, 2000] y [López and Crespo, 2003], se ha introducido el concepto de métodos genéricos, permitiendo que un método también tenga parámetros formales en su declaración.

Dentro del conjunto de tipos obtenidos a partir de las instanciaciones genéricas se distinguen dos subconjuntos, tipos completos o completamente instanciados y los tipos no completos o no completamente instanciados:

- Los tipos completos, son aquellos cuyo conjunto de parámetros reales no contienen ningún parámetro formal (`FormalPar`).

- Los tipos no completos, son aquellos cuyo conjunto de parámetros reales contienen al menos un parámetro formal (`FormalPar`).

De la definición anterior se deduce que los tipos procedentes de instanciaciones genéricas no completas están contenidos dentro de clases o métodos genéricos, ya que utilizan un parámetro formal en su instanciación.

Para identificar los tipos de forma única, procedentes de instanciaciones genéricas completas, es necesario el nombre de la clase genérica junto con la lista de los tipos de los parámetros reales que contiene la instanciación. Estos nuevos tipos tendrán un ámbito global.

Por otro lado, para identificar los tipos de forma única procedentes de instanciaciones genéricas no completas, se necesita el nombre de la clase, la lista de los tipos de los parámetros reales que contiene la instanciación y el nombre de la clase o método genérico que contiene la instanciación. Estos tipos tendrán un ámbito local donde se produce la instanciación ya que tienen una dependencia sobre los parámetros formales de la clase genérica o método que los contiene. Los parámetros formales sólo son visibles en el ámbito de la clase o método genérico donde son definidos, por tanto para identificarlos se utiliza el nombre de la clase genérica o método añadiendo el nombre del parámetro formal.

Por ejemplo, sea $C[F]$ una clase genérica con parámetro formal F, se identifica de forma única al parámetro formal F por $C@F$. Dadas dos clases genéricas $C_1[F_1]$ con parámetro formal F_1 y $C_2[F_2]$ con parámetro formal F_2 y una clase no genérica D:

- Un tipo procedente de una instanciación genérica completa $C_1[D]$, se identifica unívocamente por $C_1[D]$.

- Un tipo procedente de una instanciación genérica no completa $C_1[F_2]$ contenida dentro de la clase C_2, se identifica unívocamente por $C_1[C_2@F_2]$.

Ambas instanciaciones $C_1[D]$ y $C_1[C_2@F_2]$ son nuevos tipos de datos identificados, que pueden usarse como parámetros reales en una nueva instanciación genérica de tipo.

Dado que se permite la declaración recursiva, se pueden tener instanciaciones de la forma:

- Una instanciación genérica completa $C_1[C_1[D]]$ se identifica unívocamente por $C_1[C_1[D]]$.

- Una instanciación genérica no completa $C_1[C_1[F_2]]$ contenida dentro de la clase $C_2[F_2]$, se identifica unívocamente $C_1[C_1[C_2@F_2]]$.

El metamodelo MOON soporta tres variantes en lo referente a la acotación de los parámetros formales: subtipado, conformidad y cláusulas tal que.

- Las variantes subtipado y conformidad (variante S - `BoundS`) se agrupan y se describen en el metamodelo a través de una sola clase (`BoundS`).

  - La diferencia fundamental entre subtipado y conformidad está en el núcleo del sistema de tipos. En concreto, en cuanto a la covarianza y contravarianza en tipos de retorno y argumentos en funciones. Mientras que en subtipado se permite covarianza en tipo de retorno y contravarianza en argumentos, en conformidad se permite covarianza en ambos. En el metamodelo son soportados a través del mismo elemento estructural.
  - La acotación F se incluye en el metamodelo como un caso especial de subtipado. Se elimina toda restricción a que un parámetro formal no pueda estar acotado por tipos no completamente instanciados que contengan ese mismo parámetro formal, permitiéndose declaraciones recursivas.

- La variante cláusulas *tal que* o *where clauses* (variante W - `BoundW`) se incluye en el metamodelo a través de la clase `BoundW`, como un conjunto de propiedades con sus correspondientes signaturas.

Se debe señalar que si bien utilizando las reglas de transformación de [Kunert, 2008], un parámetro formal puede tener acotación por subtipado, conformidad y cláusulas *tal que* a la vez, las reglas semánticas descritas en [Crespo, 2000] establecen siempre una alternativa entre acotación con variante S o variante W, no pudiendo darse las dos a la vez. Esto se ha traducido en una herencia de ambos tipos de acotación respecto al parámetro formal, reflejando la semántica original de MOON.

Las variantes de acotación se utilizan en [Crespo, 2000] en la obtención del tipo calculado, equivalente a la obtención de la forma *flat view* propuesta en los entornos de trabajo de EIFFEL [Meyer, 2009]. Las variantes persiguen acotar las características de los parámetros formales para garantizar la corrección de tipos en las instanciaciones genéricas, determinando un conjunto de sustituciones válidas para los parámetros formales. Si un parámetro formal no está acotado, el tipo del parámetro real en una sustitución puede ser cualquiera.

Por otro lado, se ha modificado la multiplicidad en la relación `BoundS` con `ClassType` permitiendo acotaciones múltiples cuando se acota por subtipado o conformodidad. Aunque dicha consideración no estuviera incluida inicialmente en [Crespo, 2000] se puede incluir de manera natural en el metamodelo ampliando la semántica a acotación múltiple, puesto que conceptos parecidos como la herencia múltiple ya estaban incluidos en MOON desde un inicio.

En la Fig. 3.4 se representan las abstracciones identificadas en MOON y sus relaciones respecto a la genericidad. La asociación `genericinstantiation` entre una clase y varios tipos sólo será válida cuando la clase sea genérica. Igualmente una clase tendrá parámetros formales (`formalparameters`) si y sólo si la clase ha sido declarada como genérica.

3.1.6. Herencia

La definición de una clase en MOON puede heredar directamente de una o varias clases (herencia múltiple). Esta característica se representa a través de la asociación con multiplicidad

entre la definición de la clase (`ClassDef`) y las cláusulas de herencia (`InheritanceClause`) (ver Fig. 3.5).

Los modificadores permitidos en MOON por las reglas de herencia (`Rename`, `Redefine`, `MakeEffective` y `MakeDeferred`), dictan que una propiedad que se hereda se puede renombrar, redefinir, hacer concreta si era abstracta y hacer abstracta si era concreta respectivamente [Meyer, 1997, Meyer, 2009]. El modificador `Select` es necesario para permitir la resolución de un conflicto de selección en ligadura tardía en presencia de herencia múltiple.

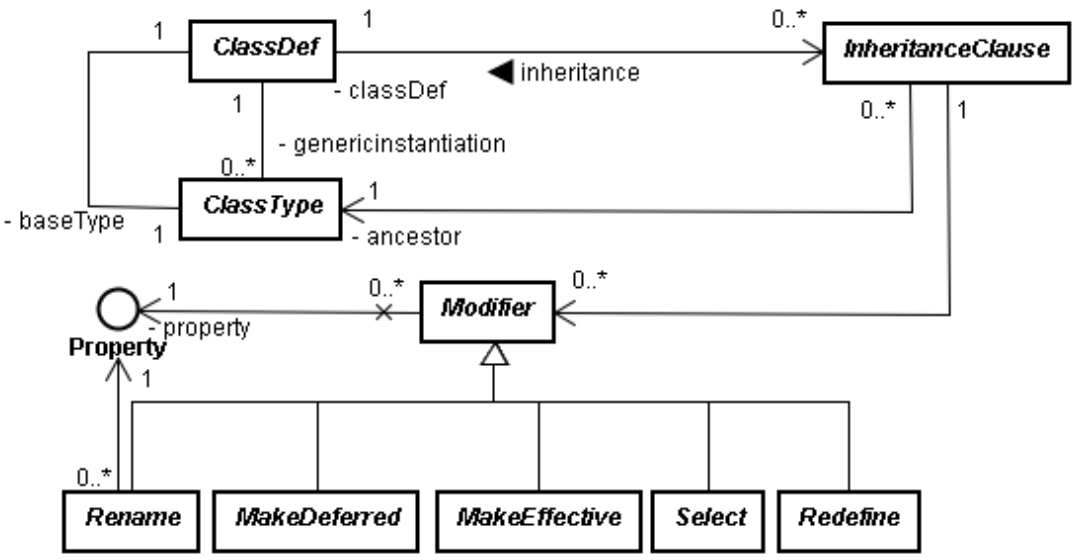

Figura 3.5: Herencia

La covarianza y contravarianza en especialización es una forma de polimorfismo múltiple soportada por algunos LPOO que permite redefinir métodos variando los tipos de los argumentos formales y valor de retorno (`Result`). En MOON se definen dos variantes en las reglas de tipos, subtipado y conformidad. En la variante de subtipado se permiten redefiniciones siempre que se respete contravarianza en argumentos formales (`FormalArgument`) y covarianza en el valor de retorno. En la variante por conformidad, la redefinición permite covarianza en argumentos formales y en el valor de retorno.

3.1.7. Expresiones e instrucciones

Con el objetivo de poder definir y analizar las refactorizaciones se hace necesario representar la información relacionada con la colaboración de los objetos a través del envío de mensajes incluidos en la implementación de los métodos. Estas colaboraciones se llevan a cabo a través

de las intrucciones que conforman el cuerpo de los métodos, así como en las expresiones utilizadas en dichas instrucciones.

En MOON, los métodos correspondientes a mensajes que solicitan servicios son rutinas, mientras que los mensajes que solicitan información se corresponden con atributos y métodos definidos como funciones. Todo envío de mensaje está asociado a una entidad:

- si el mensaje se corresponde con un atributo o una función el envío es una expresión (`CallExpr`) (ver Fig. 3.6).
- si se corresponde con una rutina es una instrucción (`CallInstr`) (ver Fig. 3.8).

Una expresión es una construcción denotando un acceso a memoria o cómputo en tiempo de ejecución que retorna un valor o referencia a objeto:

- a través de entidades de signatura, internas o predefinidas (ligadas a través de la clase `CallExpr`)).
- o bien a través de constantes manifiestas (`ManifestConstant`) (ver Fig. 3.7).

En la gramática MOON las expresiones sólo se podían invocar sobre una entidad, y no sobre una expresión. Es decir, en la expresión `e.expr`, la parte izquierda `e` se restringía a ser una entidad. Una generalización de esto consiste en asumir que la parte izquierda `e` es una expresión. Esto introduce una ambigüedad en la gramática, razón por la que se optó por la restricción. Sin embargo, al aplicar las reglas de transformación para obtener el metamodelo a partir de la gramática, se puede obtener la generalización en el metamodelo que permite que en las expresiones las invocaciones se puedan realizar tanto sobre entidades como sobre constantes manifiestas.

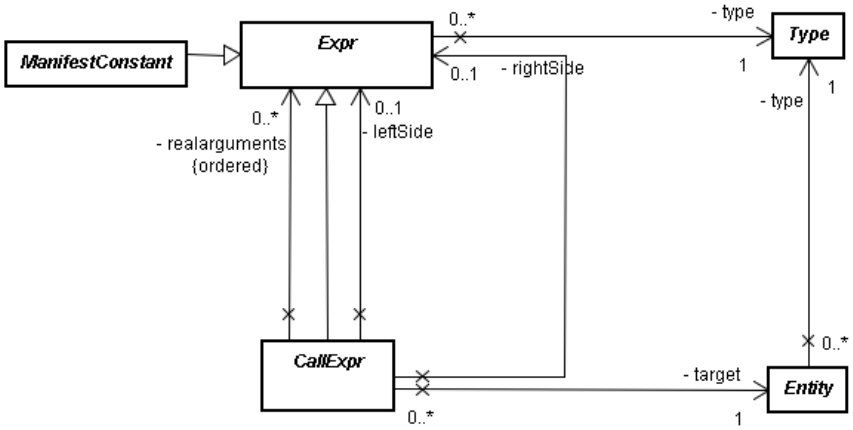

Figura 3.6: Expresiones

En [Crespo, 2000] se definió la longitud de un envío de mensajes e_m como la cantidad de llamadas encadenadas, que forman la construcción, al margen de los argumentos reales que cada llamada pueda tener, y se denota por $L(e_m)$. Por lo tanto, siendo $e_m = e_1.e_2....e_n$, $n >= 1$, la estructura de un envío de mensajes (al margen de los argumentos reales) entonces $L(e_m) = n$.

Puesto que un envío de mensajes es una forma especial de expresión, se puede hablar de longitud de una expresión, añadiendo a la definición las expresiones que se corresponden con constantes manifiestas (`ManifestConstant`) donde $L(e_c) = 1$ (ver Fig. 3.7).

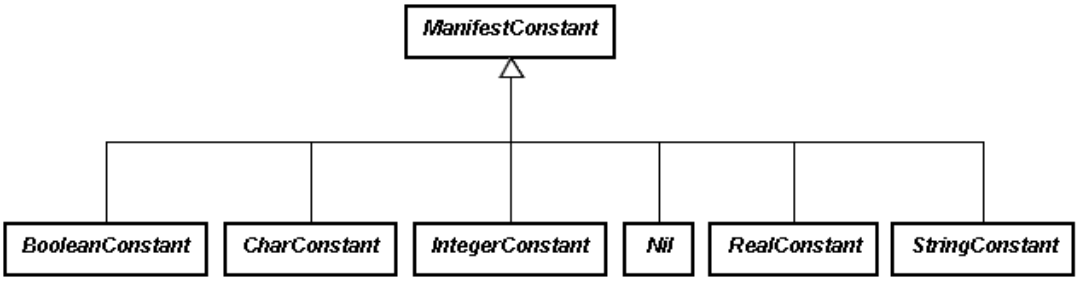

Figura 3.7: Constantes

En MOON se consideran las siguientes instrucciones:

- creación de objetos o invocación a constructores (`CreationInstr`).
- asignación (`AssigmentInstr`).
- envío de mensajes con petición de servicio a rutina (`CallInstr`).
- instrucciones compuestas o bloques de instrucciones (`CompoundInstr`).

La representación completa de cada tipo de instrucción puede observarse en la Fig. 3.8

En relación con dichas instrucciones, y dada la complejidad inicial del juego de instrucciones que se pueden extraer de los lenguajes de programación actuales, en MOON [Crespo, 2000] se consideraron las siguientes simplificaciones respecto a instrucciones y expresiones:

1. Eliminación de envíos de mensaje en cascada. Cualquiera que sea la llamada en cascada $e_1.e_2....e_n$ con $n > 2$, al margen de los argumentos que pueda tener cada mensaje involucrado en la cascada, se puede reducir considerando la siguiente secuencia de instrucciones:

   $t_1 := e_1.e_2;$

   $t_2 := t_1.e_3;$

   ...

$t_{i-1} := t_{i-2}.e_i$;

$(2 < i < n)$

si $e_1.e_2....e_n$ es una expresión, cuando $i = n$ se tendrá $t_{n-1} := t_{n-2}.e_n$;

si $e_1.e_2....e_n$ es una instrucción, cuando $i = n$ se tendrá $t_{n-2}.e_n$;

Al considerar esta simplificación y la entidad predefinida `Self` en MOON, el envío de mensajes se clasifica en envíos de longitud 1 si la entidad asociada al envío de mensaje es `Self` de forma implícita y envío de mensajes de longitud 2 en caso contrario.

2. Eliminación de algunas expresiones. En MOON no se incluyen expresiones binarias ni unarias, asumiendo que se pueden reescribir como una expresión de envió de mensajes. Por ejemplo, `a + b` deberá reescribirse como `a.+(b)`.

3. Eliminación de instrucciones de control de flujo de ejecución.

Debido a la simplificación de eliminación de envío de mensajes en cascada expuesta anteriormente los argumentos reales que puede recibir una expresión se corresponden con expresiones atómicas respetando la Ley de Demeter [Lieberherr et al., 1988, Lieberherr and Holland, 1989]. Dada una expresión $f(a.b)$ se puede representar con la siguiente descomposición de sentencias $t_1 = a.b$ y $f(t_1)$, siendo toda expresión de longitud 1 o 2.

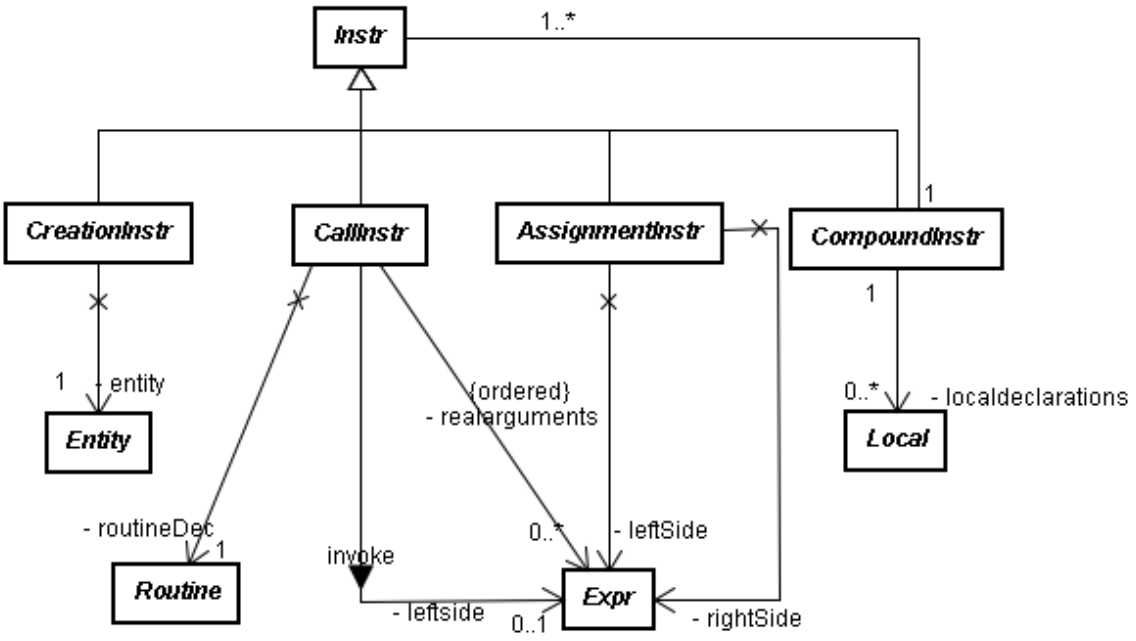

Figura 3.8: Instrucciones

En relación con las simplificaciones inicialmente expuestas en MOON, se ha tomado como solución modelar toda expresión como atómica. Por ejemplo, si se tiene una expresión `a.b.c`, se permite que una expresión sea calculada a partir de otra (formando su lado derecho, en el ejemplo `c` sería la expresión al lado derecho de `b`) o bien que a partir de una expresión se calcule otra expresión (formando su lado izquierdo, en el ejemplo `a` sería la expresión al lado izquierdo de `b`).

Como se puede ver en la Fig. 3.9, mientras que la simplificación planteada con la gramática de MOON [Crespo, 2000] introduce un conjunto de variables temporales para representar la cascada de invocaciones, como solución de modelado se ha optado por permitir que una expresión tenga otras expresiones a su lado izquierdo y derecho.

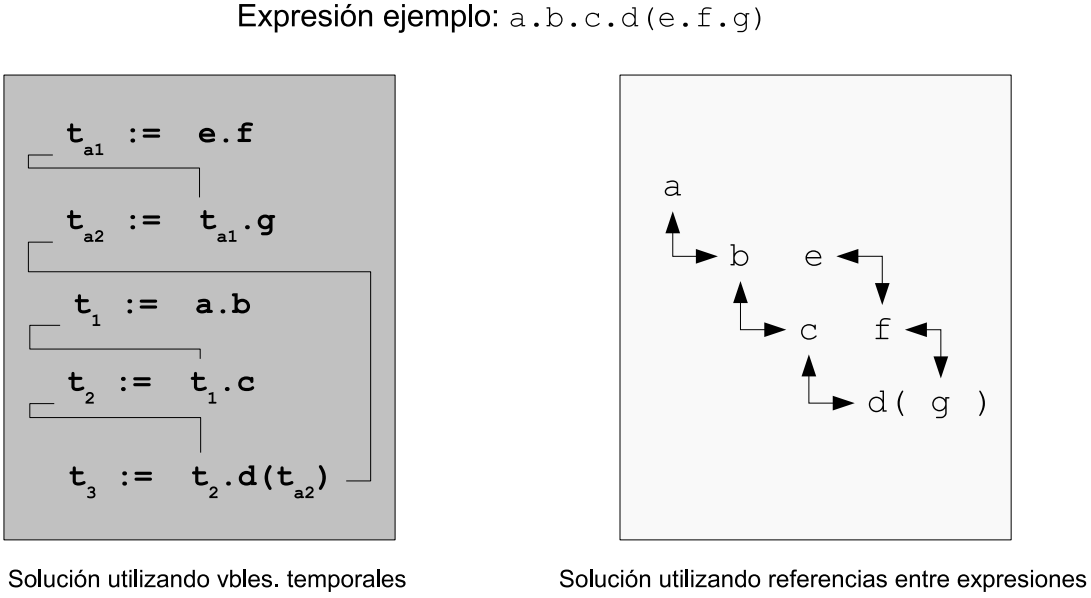

Figura 3.9: Soluciones en las invocaciones con expresiones

Se ha de considerar que en la solución inicial, se utilizaba una representación intermedia del código para facilitar su análisis en cuanto a tipos y uso de propiedades, en relación al paso de mensajes. El código original y el código refactorizado, no deben ceñirse a dicha regla puesto que las llamadas de longitud mayor de 1 son y deben ser recuperadas al volver hacia una representación con el lenguaje concreto de partida. Difícilmente se aceptaría recuperar un código equivalente con un número de variables temporales que hacen dicho código difícil de interpretar y mantener.

En la solución propuesta, el análisis del código, sigue pudiéndose realizar, vinculado a procesar recursivamente las relaciones reflexivas de una expresión, evitando la creación de un número elevado de variables temporales. Por otro lado, se elimina tanto el concepto de expresiones de longitud 2 como las invocaciones de instrucciones de longitud 2, inicialmente presentadas en [López and Crespo, 2003], simplificando en gran medida el metamodelo. Las

instrucciones de longitud mayor de 1 son invocadas (asociación `invoke` en la Fig. 3.8) sobre una expresión (que a su vez puede ser invocada sobre otra expresión recursivamente como se refleja en la Fig. 3.6).

Las simplificaciones respecto a la eliminación de instrucciones de control de flujo y operaciones unarias y binarias se mantienen, modelando las operaciones como expresiones de envío de mensaje.

Finalmente en la gramática MOON, en cuanto a las instrucciones de asignación, nos encontramos que en el lado izquierdo de una asignación sólo se permiten variables. De nuevo aquí se generaliza, al transformar de la gramática al metamodelo, con la regla semántica adicional de uso de expresiones como parte izquierda, siempre y cuando estemos ante un *l-value* tal y como se define [Aho et al., 2006] (*e.g.* en asignaciones cuyo lado izquierdo es un acceso indexado a un *array*).

Aún entendiendo que estos cambios tienen un alto impacto respecto a trabajos iniciales (y en particular respecto a la gramática de partida) se han mostrado en la práctica como simplificaciones que permite un correcto manejo y extensión del metamodelo MOON manteniendo su semántica como requisito indispensable [Kunert, 2008].

3.1.8. Comentarios de código

Una parte fundamental del código fuente, desde un punto de vista de la comprensión y mantenimiento del mismo, es la documentación aportada a través de comentarios en el código fuente. Aunque dichos comentarios sean ignorados por el compilador, son vitales para las personas que modifican el código y para las herramientas que procesan el código para generar documentación, detectar defectos, sugerir propuestas de refactorización, etc.

Aunque en el lenguaje MOON no se ha incluido inicialmente este concepto, se ha decidido su inclusión final en el metamodelo MOON dado su objetivo final de dar soporte a la construcción de refactorizaciones y su inclusión en la totalidad de LPOO. El concepto abstracto se refleja en una clase `Comment` descendiente directa de `ObjectMoon`. Su inclusión en la jerarquía principal de herencia se muestra en el Apartado 3.1.9, en la Fig. 3.10.

3.1.9. Jerarquía principal de MOON

La jerarquía del metamodelo MOON clasifica los conceptos que han sido considerados básicos y relevantes para poder llevar a cabo refactorizaciones en LPOO, estáticamente tipados y con soporte de genericidad, añadiendo unas propiedades básicas comunes a los mismos. Dicha jerarquía no se deduce directamente de la gramática de MOON, sino de la semántica de los objetos MOON y con el objetivo claro de construir un *framework* extensible basado en el metamodelo MOON.

Como se expone en [Meyer, 2009], la herencia permite definir una jerarquía de clases donde cada elemento puede tener un papel. Dentro de esa jerarquía es necesaria una clase que juegue el papel de clase universal con un propósito general, donde se definen las propiedades inherentes a todos los elementos del metamodelo. Este tipo de jerarquías está presente en la

práctica totalidad de LPOO como SMALLTALK, EIFFEL, JAVA, C#, etc. (con la excepción de C++).

En nuestro metamodelo esa clase es `ObjectMoon`. Todo elemento a considerar en una refactorización hereda de la clase `ObjectMoon` como objeto universal o raíz. Todos los descendientes de esta clase contienen un identificador único (`id`). También se almacena la información correspondiente al número de línea y columna donde aparece el objeto, respecto a su posición inicial en el código fuente, o bien valores negativos en caso de no existir referencia posible, como ocurre por ejemplo con la información extraída de ficheros compilados o binarios.

Además esta clasificación inicial no está cerrada, permitiendo su evolución a través de los mecanismos de herencia, en el caso de tener que incorporar nuevos conceptos (ver Fig. 3.10), siempre y cuando se justifique la generalidad del nuevo concepto introducido. Siempre con el objetivo fundamental de mantener una representación minimal para una familia amplia de lenguajes.

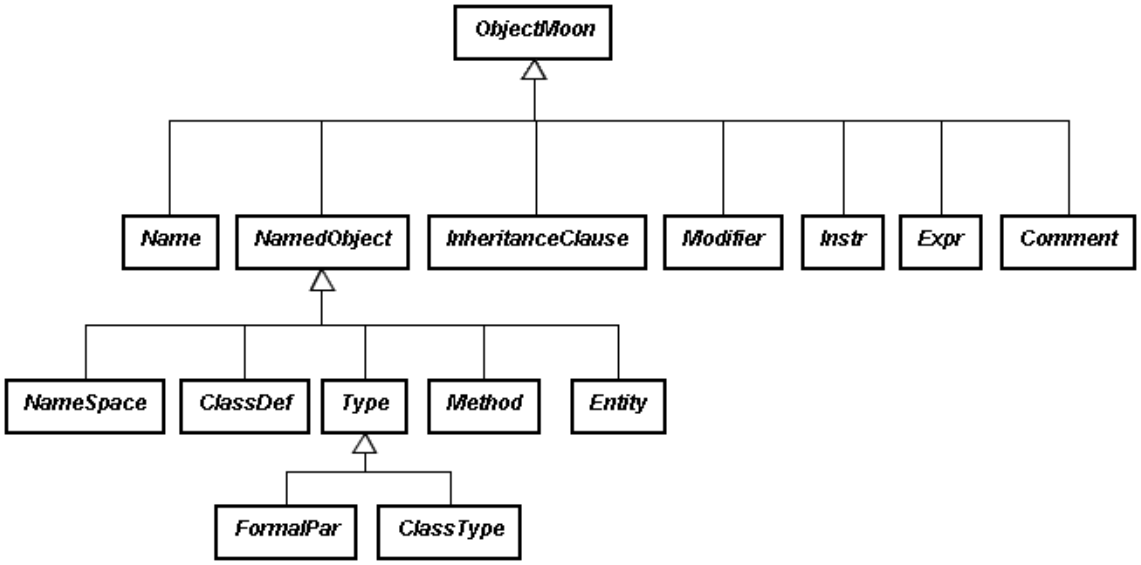

Figura 3.10: Jerarquía principal

En las refactorizaciones es necesario conocer las dependencias existentes entre los elementos representados, para poder determinar su viabilidad. Estas dependencias vienen dadas por relaciones de tipos, genericidad, herencia, asociaciones y relaciones de dependencia, derivadas a partir de las clases en la Fig. 3.10 y detalladas en secciones previas.

- Las relaciones entre tipos y clases, y su organización lógica en contextos o espacios de nombres, se derivan a partir de las clases `NameSpace`, `ClassDef` y `Type` como elementos fundamentales.

- Las relaciones de genericidad se reflejan en la introducción de parámetros de tipo `FormalPar` y características adicionales sobre clases y tipos generados por las clases `ClassDef` y `ClassType` respectivamente.

- La relaciones de herencia entre clases y tipos son representadas a través de las clases `InheritanceClause`, así como sus modificadores de herencia a través de la clase `Modifier`.

- Las entidades, con sus tipos asociados y las signaturas de métodos, determinan las operaciones a realizar a través de otras entidades, y se corresponden con las clases `Entity` y `Method`.

- Las instrucciones `Instr` y las expresiones `Expr` dan lugar a relaciones de dependencia, a través del cuerpo de los métodos, poniendo en combinación todos los elementos previos.

Dentro de esta jerarquía, se distinguen un conjunto de elementos con nombre (`NamedObject`) que se caracterizan por poseer un nombre simple (`Name`) y un nombre único calculado a partir del nombre simple y del nombre del contexto en el que se encuentran. El nombre simple sirve para identificar al elemento dentro de un contexto y el nombre único sirve para identificar al elemento, de forma única en el sistema.

Frente a versiones anteriores [López and Crespo, 2003], se ha optado por mantener como valor calculado el nombre único por ofrecer múltiples ventajas, básicamente en las actualizaciones, frente a la solución de almacenar el valor. Por otro lado se ha introducido el concepto de nombre (`Name`) para mantener una mayor uniformidad en el metamodelo, en lugar de utilizar un tipo de datos vinculado al lenguaje particular de implementación del mismo.

Todas las clases tienen su correspondencia uno a uno con elementos no terminales de la gramática, salvo `ObjectMoon`, `Name`, `NamedObject` y `NameSpace`, cuya inclusión ha sido razonada por motivos semánticos.

3.1.10. Conclusiones al metamodelo MOON

Partiendo de un lenguaje minimal, con 52 reglas de producción en su gramática, se ha llegado a un metamodelo final que consta de 50 clases e interfaces, manteniéndose en un número medio aceptable. El metamodelo recoge los elementos claves para recoger la semántica de una familia extensa de LPOO estáticamente tipados, con un número manejable de clases que permiten su extensión a lenguajes concretos de la corriente principal pudiendo abstraer cuestiones relativas a la refactorización del código en muchas facetas: tipos, herencia y fundamentalmente en este trabajo, genericidad.

En relación a los metamodelos vistos en el Capítulo 2, debemos señalar algunas conclusiones al realizar una comparativa entre su propuesta y la presentada con el metamodelo MOON.

Ventajas de Moose/ FAMIX

- El modelo de clases es más sencillo con los problemas asociados de dar un soporte limitado al concepto de tipos, y características inherentes como la genericidad.
- Inclusión de distintos formatos de intercambio (CDIF, MSE, XMI) para interactuar con un gran número de herramientas.

Desventajas de Moose/ FAMIX

- Pérdida de información, que impide trabajar con herencia avanzada y genericidad con independencia del lenguaje.
- Pérdida de información, en lo relativo al cuerpo de lo métodos, dificultando el enfoque independiente del lenguaje en un gran número de refactorizaciones.
- Implementaciones de los mecanismos de las refactorizaciones específicas para cada LPOO .

Ventajas de GrammyUML

- El modelo de clases es más sencillo, integrado con el metamodelo de UML 1.4 a partir de sólo 8 modificaciones.
- Expresividad de los contratos de las refactorizaciones con OCL.
- Futura integración con MDA.

Desventajas de GrammyUML

- Planteamiento teórico, sólo sobre dos refactorizaciones.
- Sin detallar la integración real de los LPOO al metamodelo.
- Sin revisiones del metamodelo desde [Van Gorp et al., 2003a].

Ventajas de MOON

- Se soporta el concepto de herencia avanzada: herencia múltiple, renombrados, selección, etc.
- Se soporta el concepto de genericidad al trabajar con modelos de tipos y variantes de acotación.
- Se soportan conceptos propios del lenguaje a través de extensiones del modelo MOON de forma natural sin existir saltos entre la solución independiente del lenguaje y la solución particular.

- Se mantiene la información del código fuente al incluir información de las instrucciones, no sólo el acceso a atributos o invocación de métodos, permitiendo análisis y transformaciones más complejas. Esta ventaja y la siguiente se comprenden mejor una vez que se presente el *framework* basado en MOON en el Capítulo 6.
- Todo el trabajo se realiza sobre las instancias del clases del metamodelo y sus extensiones. No se vuelve a trabajar sobre el código fuente. Con esto se elimina la impedancia entre la parte de comprobación y la parte de transformación. También se evita tener que volver a analizar el código transformado para actualizar el modelo.

Desventajas de MOON

- El número medio de clases en el metamodelo es alto.

Problemas comunes a los metamodelos

- El esfuerzo de crear un frontal o *front-end* de transformación particular del lenguaje en Moose/ FAMIX, en GrammyUML, o del parser y extensión del lenguaje en MOON conllevan un alto esfuerzo, dependiendo del nivel de detalle a alcanzar.
- La escalabilidad del sistema ante el tamaño del grafo puede verse afectada. Este problema es más acusado en MOON y sus extensiones.
- Ante la incorporación de nuevos lenguajes, el proceso obliga a un nuevo análisis de la refactorización y construcción de la instanciación o *plug-in*.

Como se puede deducir, ninguna de las soluciones basadas en metamodelos está exenta de problemas, pero desde nuestro punto de vista, la solución basada en MOON afronta con mejores garantías ciertas debilidades detectadas y satisface algunas necesidades no cubiertas por otros metamodelos. La validación de su adecuación como soporte de lenguajes concretos para definir refactorizaciones, se desarrolla con más detalle en el Capítulo 7.

3.2. Arquitectura general

En la Fig. 3.11, se muestra la arquitectura general que será la base inicial del presente trabajo. Partiendo del análisis de la información obtenida del código fuente (y código binario asociado) para un lenguaje objetivo concreto, se representará dicha información sobre un metamodelo particular para el lenguaje objetivo inicial. Este metamodelo particular debe extender y cumplir las reglas marcadas en el metamodelo MOON, detallado previamente. La evolución de dicha propuesta se encuentra recogida en distintos trabajos como [López et al., 2004, Marticorena, 2005, Marticorena et al., 2007b].

Una vez que la información ha sido extraída y almacenada en instancias del metamodelo correspondiente, se podrán ejecutar las refactorizaciones. Los distintos elementos de las

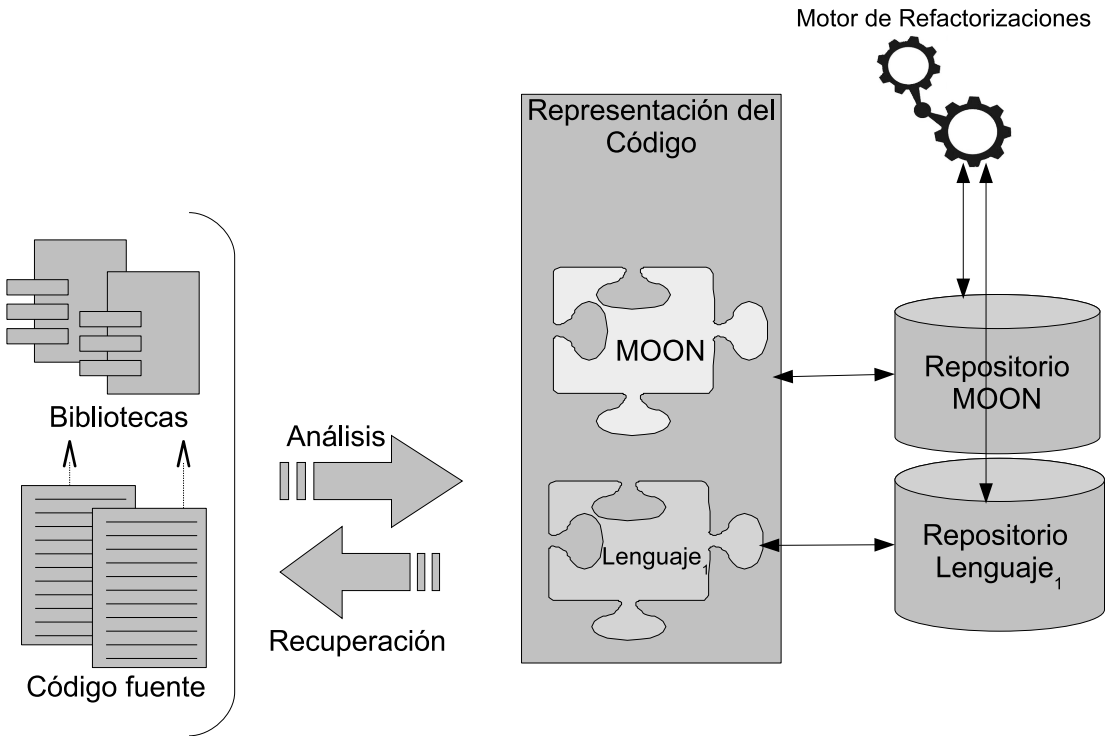

Figura 3.11: Arquitectura general

refactorizaciones (consultas y acciones) se definirán sobre el metamodelo MOON o sobre el metamodelo del lenguaje utilizado dependiendo del nivel de independencia obtenido. Para ello se definirán e implementarán las refactorizaciones y sus elementos concretos, en lo que se denominarán repositorios vinculados a MOON o al lenguaje objetivo concreto.

En una situación ideal, la refactorización utilizaría sólo elementos del repositorio MOON, y por lo tanto sería completamente independiente del lenguaje. Por el contrario, cuantos más elementos de la refactorización sean tomados del repositorio para el lenguaje concreto, menor grado de independencia se logra en la refactorización.

Una vez aplicada la refactorización por el motor de refactorizaciones, la representación del código inicial habrá sido modificada, cambiando su estado. Para obtener el código refactorizado es necesario un proceso de recuperación de código a partir de dicha representación, de nuevo trabajando sólo sobre el metamodelo del lenguaje concreto.

En el conjunto completo de la arquitectura, se aplicará el concepto de *framework* orientado a objetos [Fayad et al., 1999]. Para ello se define y construye la parte operativa de la refactorización a un alto nivel, planteando de forma muy abstracta el soporte de su ejecución sobre el motor de refactorizaciones, que transforma las instancias del código a refactorizar (idealmente instancias de MOON).

El concepto de inversión de control presente en un *framework* se adapta perfectamente a esta arquitectura propuesta. Los predicados y acciones concretos de las refactorizaciones, así como los objetos que representan el código escrito en el lenguaje objetivo, son conectados y puestos en ejecución como extensiones de dicho *framework* (tanto como elementos del motor de refactorizaciones como extensiones al metamodelo MOON). Esto evita repetir esfuerzos de implementación, reutilizando en gran medida la parte funcional del *framework*.

Puesto que además la arquitectura planteada ofrece otro conjunto de posibilidades [Crespo et al., 2006a], es necesario clarificar cuáles NO son objetivo de la presente tesis:

Detectar oportunidades de refactorización aunque en la propuesta de las refactorizaciones se sugieren motivos para su aplicación, en su mayoría presentadas de forma textual, no se pretende dar en este trabajo una base a la detección basada en diferentes técnicas (métricas, heurísticas, evolución del proyecto, etc.). Aunque es viable afrontar dicha detección desde las bases aquí planteadas, y en especial desde enfoques más orientados a la independencia del lenguaje [Crespo et al., 2005a, Crespo et al., 2005b, Crespo et al., 2006a], quedan fuera del ámbito del presente trabajo.

Refactorizaciones basadas en análisis del flujo del programa Aunque el resultado de observar la ejecución de un programa pueda sugerir ciertos problemas, no será objetivo de las refactorizaciones aquí tratadas. En esta misma línea, refactorizaciones con el objetivo de optimizar rendimientos no son tampoco tema del presente trabajo. Este tipo de transformaciones exigiría una discusión sobre el posible conflicto entre la mejora del rendimiento frente al incremento de la dificultad de la comprensión y facilidad de mantenimiento del código.

Observación y medida de la mejora de la calidad En la mayoría de catálogos se deduce que la aplicación de la refactorización mejora el diseño actual [Fowler, 1999]. Sin embargo no se establece un proceso de medida, para verificar que efectivamente esto se consigue. En este trabajo no se establece un proceso con tal fin, aunque queda abierta la posibilidad de completar el proceso de refactorización que se establezca en este trabajo, con otras tareas que realimenten y refinen el proceso.

En base a todo lo expuesto en este capítulo, se establece la arquitectura y la hipótesis general de trabajo, enmarcadas en el ámbito de una búsqueda de reutilización, a través del mayor grado de independencia del lenguaje al aplicar refactorizaciones, particularmente en genericidad, utilizando el metamodelo MOON.

CAPÍTULO 4

DEFINICIÓN, CARACTERIZACIÓN Y PROCESO DE CONSTRUCCIÓN DE REFACTORIZACIONES

Una vez establecidos el contexto de partida en el Capítulo 2 y la base fundamental para abordar el problema de la independencia del lenguaje, con el metamodelo MOON, en el Capítulo 3, en este capítulo nos centraremos en el estudio de las refactorizaciones y en la propuesta de soluciones a aquellos aspectos que no han sido tratados en profundidad.

En particular, en la Sec. 4.1 se establecerá la estructura en la definición de refactorizaciones que se utilizará a lo largo del presente trabajo . Con esta estructura se definirá el catálogo de refactorizaciones descrito en el Capítulo 5.

En la Sec. 4.2 se define una caracterización de refactorizaciones que permite clasificar tanto las refactorizaciones ya conocidas, como las que se proponen en el Capítulo 5. La caracterización presta especial interés a características relevantes de cara a la incorporación de refactorizaciones en herramientas y entornos de desarrollo integrados.

Finalmente a partir de dicha caracterización, se describe en la Sec. 4.3 una propuesta de proceso para la definición y construcción de refactorizaciones.

4.1. Estructura en la definición de refactorizaciones

Mientras que el concepto de lo que es una refactorización está sólidamente establecido (ver Capítulo 2), existen multitud de variantes en la estructura y formalización a utilizar para su definición. Es por lo tanto necesario establecer los elementos que componen una refactoriza-

ción, así como indicar los formalismos a utilizar para el posterior análisis e implementación de refactorizaciones en los siguientes capítulos.

En el estado del arte de la definición de refactorizaciones pueden observarse distintas líneas en los elementos que las componen. En general, se establece que una refactorización tiene que estar definida por dos elementos básicos: las condiciones que hacen efectiva su aplicación, denominadas generalmente como precondiciones, y las posteriores transformaciones realizadas.

La correcta definición de las precondiciones es un punto fundamental, y también problemático, puesto que su verificación debe garantizar la preservación del comportamiento, intrínseca a la aplicación de una refactorización. La principal dificultad estriba en definir el conjunto de precondiciones minimal que asegure esta propiedad y, por otra parte, no impida la transformación en casos en los que sí podría ser aplicable.

Como se indicó en el Apartado 2.1.7, tomando como ejemplo una refactorización como **Eliminar Clase**, cada autor daba un conjunto diferente de precondiciones. Incluso dándose el caso de que en [Fowler, 1999, Fowler, 2013], no se llega a definir esta refactorización.

Como se pudo observar de este ejemplo, autores que abogan por elementos similares en la definición y estructura de las refactorizaciones, establecen distintos nombres y precondiciones en refactorizaciones que intuitivamente son la misma. En el ejemplo planteado la primera definición [Opdyke, 1992, pág.41] establece criterios más pesimistas, mientras que la segunda solución [Roberts, 1999b, pág.27], amplía las posibilidades de refactorización. En el tercer caso [Fowler, 1999, Fowler, 2013], ni siquiera se ha considerado su inclusión en el catálogo de refactorizaciones.

Además cabe plantearse si las diferencias pueden deberse al hecho de que la refactorización esté definida para lenguajes objetivo diferentes en cada uno de los casos (C++, Smalltalk y Java respectivamente). Esto también se acusa al establecer las transformaciones a realizar sobre el código.

En el Apartado 4.1.1 se muestran los elementos de las refactorizaciones identificados por distintos autores. Posteriormente tras su análisis en el Apartado 4.1.2, se presenta la propuesta que se seguirá a lo largo del presente trabajo.

4.1.1. Estado del arte en la estructura de la definición de refactorizaciones

En los trabajos de [Opdyke, 1992] se hace especial hincapié en el uso de invariantes y precondiciones para la verificación de la semántica estática y dinámica, utilizando lógica de predicados de primer orden.

En dicho trabajo se describían siete propiedades básicas (invariantes), descritas previamente en el Apartado 2.1.4, que debían ser mantenidas por las refactorizaciones para asegurar la preservación del comportamiento. Sin embargo, no se consideran los puntos de variabilidad que permitan extender la solución a otros lenguajes.

La estructura propuesta en [Opdyke, 1992] tiene los siguientes elementos.

Nombre	Establece un vocabulario común.
Acciones	Transformaciones a realizar sobre el código.
Argumentos	Entradas objetivo de la refactorización.
Efectos	Resultados observables de la aplicación de las transformaciones.
Precondiciones	Condiciones que deben cumplirse para asegurar la preservación del comportamiento.
Preservación del comportamiento	Razonamientos sobre la preservación del comportamiento como resultado de la refactorización.

Aunque los razonamientos no se expresan con formalismos, su intención es asegurar que la ejecución de las refactorizaciones mantiene el mismo funcionamiento del software refactorizado antes y después de la refactorización.

A continuación se muestra un ejemplo, con la refactorización **CHANGE_CLASS_NAME** definida en el catálogo de [Opdyke, 1992], aplicando su estructura:

Nombre	**CHANGE_CLASS_NAME**
Acciones	Cambia el nombre de la clase.
Argumentos	clase C, nombre S
Efectos	Este cambio de nombre es reflejado a lo largo del programa (*i.e.* en clases y subclases, constructores, destructores, y la declaración/instanciación de la clase).
Precondiciones	El nuevo nombre no colisiona con el nombre de una clase ya existente $\forall class \in Program.classes, class.name \neq S$.
Preservación del comportamiento	La precondición asegura nombres de clases distintos. Cambiar el nombre de una clase no modifica su comportamiento. Otras propiedades se preservan de forma trivial.

La línea de trabajo de [Opdyke, 1992] fue continuada por [Roberts, 1999b], añadiendo el concepto de postcondiciones. Sus definiciones son semiformales, utilizando también lógica de predicados de primer orden sobre los elementos que constituyen un programa orientado a objetos. Su solución permite una composición de refactorizaciones que en la solución previa no se tenía. Dada la similitud de ambos trabajos, presentan ventajas y desventajas comunes. Ambos enfoques [Opdyke, 1992, Roberts, 1999b] son pesimistas en el sentido de que pueden impedir a priori ejecutar refactorizaciones que podrían ser correctas.

Los elementos comunes son el nombre, entradas, precondiciones y las acciones, a las cuales se añaden las postcondiciones en [Roberts, 1999b]. Las postcondiciones se deducen de los efectos de las acciones de transformación realizadas. Se define un catálogo de lo que el autor denomina funciones de análisis primitivas y derivadas para consultar, y funciones de transformación para modificar el código, expresadas en lógica de predicados de primer orden. Sin embargo, por una parte, el catálogo de funciones dado no parece completo, quedando pendiente su compleción para poder definir las refactorizaciones de forma más precisa. Por otra parte, la propia definición de la refactorización se describe informalmente dejando al lector que decida qué funciones se necesitan para formalizar cada refactorización.

En [Fowler, 1999] se proponen definiciones de refactorizaciones más informales[1], dando una lista de elementos que completan la descripción de una refactorización como una receta, al estilo de otras soluciones adoptadas en la definición de patrones de diseño. Se acompaña la descripción, con un diagrama de clases utilizando UML, mostrando una instantánea antes y después de realizar la refactorización. Este tipo de lenguajes basados en representaciones gráficas se ha extendido con la utilización de *thumbnails* [Gorts, 2008] para describir las refactorizaciones existentes. Aunque debemos señalar que pese a ser intuitivas, estas representaciones carecen de precisión.

La estructura de la definición de refactorizaciones en [Fowler, 1999] se muestra a continuación:

Nombre	Establece un vocabulario común.
Resumen	Situación en la que se necesita la refactorización y resumen de qué hace la refactorización. Permite una búsqueda más ágil de una refactorización en el catálogo.
Motivación	Explicación de por qué hay que realizar la refactorización y razones por las que NO debería ser llevada a cabo.
Mecanismos	Descripción paso a paso de cómo llevar a cabo la refactorización.
Ejemplos	Usos de la refactorización para ilustrar cómo trabaja. Se aplica para JAVA como lenguaje objetivo concreto.

La propuesta utiliza las *recetas*, aplicando secuencialmente las operaciones o pasos indicados en los *mecanismos*. La preservación del comportamiento se asegura repitiendo en algunas ocasiones la compilación (asegurando semántica estática) y la ejecución de las pruebas (asegurando la semántica dinámica). No parece existir un criterio objetivo respecto a cuándo compilar o probar, tras las distintas transformaciones realizadas.

Por otro lado, aplicando dichas recetas, es posible llegar a algún punto de la ejecución de una refactorización que nos lleve a un estado inconsistente, pero no se indica cómo se debe llevar a cabo la operación de deshacer para recuperar el código original, suponiendo que existe algún mecanismo de recuperación en el caso de que surja cualquier problema.

A continuación se da una ejemplo abreviado de la refactorización **EXTRACT CLASS (149)** [Fowler, 1999] para poder ver sus elementos:

[1] En dicha obra se denomina como *formato* de la refactorización a lo que aquí se denomina *estructura*.

Nombre	**Extract Class** (ver Fig. 4.1)
Resumen	Una clase está haciendo trabajo que debería ser hecho por dos clases.
Motivación	La clase ha crecido a lo largo del tiempo incorporando mucha funcionalidad que debe ser separada. Es necesario revisar las referencias a los elementos que van a ser separados y las relaciones de herencia para comprobar los efectos de la refactorización.
Mecanismos	1. Decidir cómo dividir las responsabilidades de la clase. 2. Crear una nueva clase para recoger las responsabilidades separadas. 3. Crear un enlace de la antigua a la nueva clase. 4. Usar **Move Field (146)** sobre cada campo a mover. 5. Compilar y probar tras cada movimiento. 6. Usar **Move Method (142)** sobre cada método a mover. Comenzar con métodos de bajo nivel hacia los de alto nivel. 7. Compilar y probar tras cada movimiento. 8. Revisar y reducir las interfaces de cada clase. 9. Decidir si se expone la nueva clase. Si se expone, decidir si se expone como un objeto referencia o como un objeto valor inmutable.
Ejemplos	código Java (se omite por brevedad).

Como se puede ver, una refactorización en [Fowler, 1999] no especifica de manera formal o semiformal las precondiciones que pueden impedir dicha refactorización. En todo caso, se puede extraer en forma textual algunas de las condiciones, pero sujetas a interpretación subjetiva.

Este enfoque es más optimista, enfrentándose posteriormente al problema de errores en compilación o en las ejecución de la batería de pruebas. Esta comprobación se puede realizar después de ejecutar algunos de los pasos descritos en los mecanismos.

En [Kerievsky, 2004] se mantiene una estructura similar a [Fowler, 1999] a la hora de definir refactorizaciones que transforman el código hacia la aplicación de patrones de diseño.

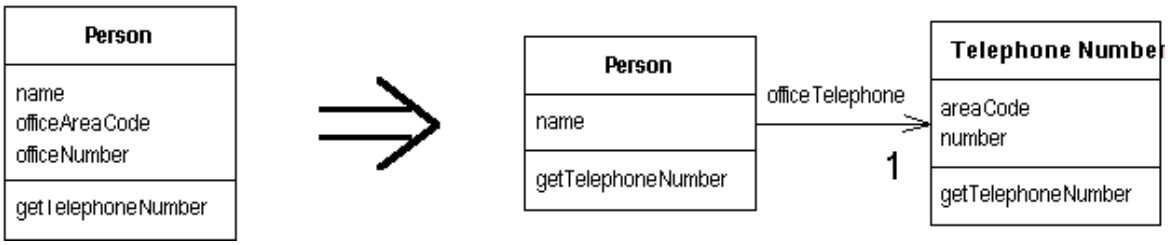

Figura 4.1: Refactorización **Extract Class**

A continuación se da una ejemplo abreviado de la refactorización **Replace Implicit Tree with Composite** [Kerievsky, 2004] para poder ver sus elementos:

Nombre	**Replace Implicit Tree with Composite**
Resumen	Implícitamente se crea un árbol utilizando una representación primitiva, como por ejemplo un `String`. Se reemplaza el tipo primitivo con un **Composite**.
Motivación	Para reducir el alto acoplamiento entre el código que construye el árbol y la propia representación del árbol. El patrón **Composite** encapsula la representación del árbol. *(Se omite por brevedad la explicación completa).*
Mecanismos	1. Identificar el código a refactorizar de construcción del árbol. 2. Identificar los tipos de los nodos para el patrón **Composite**. Diseñar uno o más tipos concretos para los nodos y no preocuparse de crear un tipo abstracto de nodo (podría no necesitarse). Crear un método para validar el contenido del **Composite** básico con un primer nodo. 3. Dar a los nodos la habilidad de añadir hijos. No proporcionar la posibilidad de eliminar hijos si la aplicación sólo añade nuevos nodos. 4. Compilar y probar. 5. Si es necesario, proporcionar a los clientes un modo de establecer atributos del nodo. 6. Compilar y probar. 7. Reemplazar la construcción original del árbol por el nuevo **Composite**. 8. Compilar y probar.
Ejemplos	código Java (se omite por brevedad).

También en [Monteiro and Fernandes, 2005] se utiliza una estructura similar para definir refactorizaciones sobre aspectos. En otros contextos, como las refactorizaciones en bases de datos, también se ha mantenido una descripción textual no formal [Ambler and Sadalage, 2006].

Otra línea complementaria en la definición de refactorizaciones es traducir la estructura de precondiciones, postcondiciones, invariantes y transformaciones al concepto de transformaciones de grafos [Mens and Lanza, 2002]. En esta línea, los programas se expresan como *grafos*, las refactorizaciones se corresponden con *reglas de producción del grafo*, y las pre/postcondiciones de la refactorización, como *pre y poscondiciones de aplicación*. Aunque utilizando otro formalismo diferente en la definición de las refactorizaciones, los elementos estructurales básicamente se mantienen.

4.1.2. Una propuesta para la estructura de la definición de refactorizaciones

En trabajos previos [Marticorena and Crespo, 2003, López et al., 2003, Crespo et al., 2004] se estableció una plantilla para la definición de refactorización que mantenía ciertos elementos de las propuestas presentadas en el Apartado 4.1.1. Mientras que se mantenían algunas definiciones de elementos de forma textual, se incorporó una definición semiformal con lógica de predicados de primer orden sobre elementos básicos de una refactorización.

Como base fundamental en las definiciones, se toma el metamodelo MOON (ver Capítulo 3), con el propósito de obtener el mayor grado de independencia del lenguaje en la definición de refactorizaciones. Tal y como se detalla en la Sec. 4.3, la refactorización queda definida para MOON, y posteriormente para cada lenguaje objetivo particular se revisará el conjunto de precondiciones, acciones y postcondiciones. La estructura planteada se muestra en la Tabla. 4.1.

Tabla 4.1: Estructura propuesta en [Marticorena et al., 2003] para la definición de refactorizaciones

Nombre	Nombre identificativo.
Descripción	Descripción en líneas generales de las acciones llevadas a cabo en la refactorización.
Motivación	Motivación para iniciar la refactorización o situación inicial que lleva a la ejecución de la refactorización.
Entradas	Elementos básicos que son objetivo y dirigen las acciones de la refactorización.
Precondiciones	Condiciones que aseguran que es correcto aplicar la refactorización respecto a las propiedades a conservar.
Acciones	Acciones que modifican el estado del código.
Postcondiciones	Condiciones que aseguran el correcto estado tras la ejecución de la refactorización.

Un punto de especial interés es la motivación de la refactorización. Aunque este trabajo no se centra en la detección de oportunidades de refactorización, siempre debe justificarse la causa o razón para realizarse la transformación.

Las entradas deben determinar cuál es el objetivo concreto de la refactorización a ejecutar. Básicamente, como resultado de establecer una correcta motivación, el usuario o el sistema deben indicar las entradas que dirigen el funcionamiento del resto de elementos básicos (pre/postcondiciones y acciones).

Esta estructura puede ser aplicada a la mayoría de catálogos de refactorizaciones revisados, mejorando y completando en la mayoría de los casos las definiciones de las refactorizaciones contenidas en dichos catálogos. A continuación se presenta un ejemplo de la refac-

torización **Push Down Field** (329) [Fowler, 1999] (ver Tabla 4.2) equivalente a **Move Member Variable to Subclasses** [Opdyke, 1992], planteada desde la semántica del metamodelo MOON. La descripción completa de predicados y funciones utilizada se recogió en [Marticorena and Crespo, 2003].

Tabla 4.2: Refactorización **Push Down Field** de acuerdo a la estructura definida en [Marticorena et al., 2003]

Nombre	**Push Down Field**
Descripción	Mover el atributo a los descendientes directos
Motivación	Un atributo es sólo utilizado por alguno de los descendientes directos
Entradas	atributo (a) y clase (C)
Precondiciones	- El atributo no debe ser referenciado por otras clases ni por la propia clase: $ReferenceAttribute(a, C) = \emptyset$ - Las subclases directas no contienen un atributo con el mismo nombre (en lenguajes con ocultación de atributos): $\forall D \in DDesc(C), \nexists b \in Attributes(D)/EqualName(a, b)$
Acciones	1. $RemoveAttribute(a, C)$ 2. $\forall D \in DDesc(C) \Rightarrow AddAttribute(a, D)$
Postcondiciones	- El atributo no pertenece a la clase C: $a \notin Attributes(C)$ - El atributo es propiedad intrínseca/esencial en todos los descendientes directos de C: $\forall D \in DDesc(C)/a \in EP(D)$

Sin embargo esta forma de definición de la refactorización introduce la dificultad de validar su corrección respecto del metamodelo MOON. Por otra parte, definir de forma precisa las acciones de transformación usando una lógica de predicados introduce una distancia conceptual y una dificultad añadida.

En base a esto se propone, manteniendo la estructura propuesta para la definición, aportar una descripción precisa con precondiciones, acciones y postcondiciones mediante el uso del lenguaje Alf [OMG, 2010, OMG, 2012], para realizar una definición de la refactorización validable sobre los elementos del metamodelo MOON. La descripción detallada del uso de Alf para la definición de las refactorizaciones se desarrolla ampliamente en el Capítulo 5.

La estructura para la definición de la refactorización se fija en la descrita en este apartado. Se aprovecha alguna simplificación implícita, como por ejemplo la navegación del atributo

hacia la clase que lo contiene, permitiendo la eliminación de entradas que son calculadas. La primera parte relativa a la descripción y motivación se encuentra incluida como comentarios, mientras que el nombre y las entradas se encuentran de manera implícita en la signatura de la actividad definida para la refactorización.

La definición de cada precondición, acción y postcondición se realiza mediante una actividad que se describe en un fichero en lenguaje ALF. La definición ALF de la refactorización indica mediante comentarios el inicio de las precondiciones, acciones y postcondiciones respectivamente.

De esta forma, la refactorización **PUSH DOWN FIELD** quedaría definida en ALF como se puede ver en el Código 4.1.1.

Código 4.1.1: Refactorización **PUSH DOWN FIELD** utilizando ALF como lenguaje para su descripción

```
namespace Moon::Refactoring;

public import Moon::Predicate::ExistsAttributeWithEqualNameInDirectDescendants;
public import Moon::Predicate::ForAllDirectDescendantsIncludesAttributeAs;
public import Moon::Predicate::ReferenceAttributeEmpty;
public import Moon::Predicate::ExistsAttributeWithEqualName;

public import Moon::Action::AddFieldInSubclassesAction;
public import Moon::Action::RemoveFieldAction;

activity PushDownFieldRefactoring // Name
    // Description: Move the field to those subclasses.
    // Motivation: A field is only used by some subclasses.
    // Inputs
    (in attDec: Attribute) : Boolean
{

  ClassDef oldClassDef = attDec.getClassDef(); // save old class

  // Preconditions
  if ( ReferenceAttributeEmpty(attDec) &&
         !ExistsAttributeWithEqualNameInDirectDescendants(attDec)) {
        // Actions
    // add the field in subclasses
        AddFieldInSubclassesAction(attDec);
    // remove the field in current class
        RemoveFieldAction(attDec);
  }
  // Postconditions
  return !ExistsAttributeWithEqualName(oldClassDef, attDec)
            && ForAllDirectDescendantsIncludesAttributeAs(attDec);
}
```

Dicha refactorización depende de los predicados, `ReferenceAttributeEmpty`, `ForAllDirectDescendantsIncludesAttributeAs` y `ExistsAttributeWithEqualNameInDirectDescendants`. Las modificaciones a las instancias del metamodelo se realizan a través del concepto de acciones: en este ejemplo, a través de las acciones `AddFieldInSubclassesAction` y `RemoveFieldAction`.

La solución propuesta enriquece la definición de refactorizaciones respecto a trabajos previos, aunque también presenta algunas debilidades. Mientras que en la Sec. 5.1 se realiza un análisis más detallado de las oportunidades y amenazas particulares del uso de ALF, como lenguaje para la especificación de refactorizaciones, a continuación se enumeran sus ventajas y desventajas desde el punto de vista de la utilización la estructura propuesta con ALF como lenguaje:

Ventajas

- Semántica precisa de la definición, no pudiendo darse diferentes interpretaciones una vez fijado el metamodelo y la refactorización. Este problema es acusado en catálogos con definiciones basadas en texto.
- Reducción del desajuste entre el modelo utilizado y la definición de la refactorización. En nuestro caso concreto, el metamodelo se expresa en UML, mientras que la refactorización se expresa en ALF, un lenguaje de acciones ideado para permitir la consulta y transformación de instancias de modelos UML.
- Reducción del desajuste entre la definición de la refactorización y la solución de implementación final. Mientras que la utilización de lógica de predicados dificultaba la expresividad en cuestiones relativas a las acciones, en ALF se pueden detallar las operaciones a realizar sobre las instancias del metamodelo, tanto desde una forma declarativa como también en forma imperativa.
- La traducción desde ALF a los distintos lenguajes de programación de la corriente principal, en particular aquellos orientados a objetos, se puede realizar de una forma suave, dado que el propio lenguaje ha sido diseñado con dicho objetivo.

Desventajas

- No se asegura que el conjunto de precondiciones sea completo para su aplicación a otros lenguajes.
- El conjunto de acciones puede necesitar ser modificado cuando se aplica la refactorización sobre un lenguaje objetivo concreto.
- La preservación del comportamiento depende de lo completas que sean las precondiciones y de la correcta implementación de la acciones.
- Las postcondiciones son deducidas directamente de los resultados de las acciones, introduciendo cierta redundancia. Esto hace que estos elementos puedan ser marcados como opcionales.

La estructura propuesta en la definición de refactorizaciones y el uso de ALF como lenguaje para su especificación, facilita su posterior evolución. Como se desarrolla en la Sec. 4.3,

nuestra propuesta asume un proceso iterativo e incremental para obtener una definición más precisa.

4.2. Caracterización de refactorizaciones

Las herramientas de desarrollo implementan las operaciones de refactorización siguiendo las definiciones de los catálogos con alguna pequeña variación. Aunque estas definiciones son manejables para la comprensión del concepto, no están tan bien definidas para la automatización de las refactorizaciones.

Los catálogos organizan y caracterizan las refactorizaciones de acuerdo a distintos criterios. Por ejemplo, el propósito de la refactorización ha sido utilizado para caracterizar la operación fundamental realizada, como *componer métodos*, *mover propiedades*, *organizar datos*, *simplificar condicionales*, *simplificar la invocación a métodos* y *manipulando la generalización* en [Fowler, 1999].

Sin embargo, estas clasificaciones de refactorizaciones propuestas no están orientadas a su implementación e inclusión en herramientas tal y como se vio en la Sec. 2.1. Para avanzar en el desarrollo de herramientas que asistan el proceso de refactorización completo, es necesario disponer de caracterizaciones que permitan establecer taxonomías y clasificaciones, caracterizando las refactorizaciones respecto a la complejidad intrínseca en su definición y posterior implementación.

A través de su caracterización, se debe facilitar la selección de refactorizaciones cuando se intenta dar respuesta a cuestiones como la determinación del orden de implementación, o cuestiones relativas a la usabilidad o reutilización, de manera que se mejore el proceso de implementación de este tipo de herramientas.

Implementar herramientas de refactorización sin ninguna guía, implica razonar previamente sobre el mejor plan para su implementación. Es una cuestión importante con el fin de reutilizar los posibles resultados intermedios. La complejidad de las refactorizaciones tiene implicaciones en la planificación, la asignación de personal, los tiempos de desarrollo, etc. El diseño de las interfaces gráficas es también relevante.

Todos estos puntos deben ser analizados, pero la documentación disponible no ofrece una guía clara al respecto. Así pues los desarrolladores a menudo toman decisiones que acaban en soluciones *ad-hoc*, donde cada refactorización se implementa desde cero, dando como solución final implementaciones no reutilizables y obligando a repetir esfuerzos en la inclusión de nuevas refactorizaciones.

En concreto, con la caracterización propuesta en esta sección, se busca dar respuesta a cuestiones del tipo:

- Dado un conjunto de refactorizaciones especificadas en un catálogo, poder determinar un orden de implementación de las mismas.

- Dado un conjunto de refactorizaciones, determinar su disponibilidad a la hora de ser aplicadas en función del elemento de código seleccionado por el usuario, mejorando la usabilidad de la herramienta de refactorización.

- Dado un conjunto de refactorizaciones, asistir en general a su reutilización, y en particular en la construcción con reutilización de sus interfaces gráficas.

4.2.1. Caracterización propuesta

En este apartado se presenta un conjunto de criterios para caracterizar refactorizaciones de múltiples catálogos. La aplicación individual de cada criterio, y en mayor medida la combinación de los mismos, debe guiar la implementación de refactorizaciones. Los criterios han sido agrupados para representar las relaciones, el ámbito sobre el que actúa y el tipo de conocimiento asociado a su comprensión.

Los categorías propuestas, en un primer nivel, son lo suficientemente generales para ser extendidas a otros catálogos de refactorizaciones. Por ejemplo, si las refactorizaciones actúan sobre código sin inclusión de genericidad (como ocurre en el catálogo de [Fowler, 1999] utilizando versiones de JAVA anteriores a 1.5) entonces las entidades objeto de estudio en la caracterización serían: sistema, clases, atributos, métodos, parámetros, instrucciones, variables locales. Si se aplican sobre refactorizaciones de diagramas de estado [G. Sunyé et al., 2001], las entidades a estudiar serían: acción, actividad, estado, transición, etc.

A continuación se van a definir las categoría básicas de la caracterización propuesta. Dicha caracterización se encuentra también descrita en [Marticorena et al., 2011c]. Uno de los objetivos clave es definir y establecer propiedades simples que puedan ser deducidas de la definición de las refactorizaciones (aplicable tanto a definiciones en forma de *"recetas"*, definiciones semiformales o formales en la línea de lo expuesto en la Sec. 4.1). Los valores de estas propiedades deben ser inferidos de la forma más objetiva posible, de tal modo que diferentes personas con diferente bagaje y experiencia, den los mismos valores.

Ámbito *(Scope)*

En los LPOO se puede considerar a la clase como pieza clave del desarrollo y unidad principal. En relación al ámbito de una refactorización, parece ser el nivel de granularidad adecuado al que registrar los cambios fruto de aplicar la refactorización. El criterio propone observar los efectos de una refactorización por el usuario, sin incluir detalles de implementación, desde un punto de vista previo y posterior a su aplicación, sin entrar a analizar los mecanismos de cambio aplicados.

Cuando se aplica una refactorización, se fijarán tres niveles de cambio de menor a mayor complejidad:

- **Interno a una clase (*Intraclass* - I)**: cambios que no afectan a otras clases. Los cambios son contenidos o limitados a la clase objetivo de la refactorización. Se modifica

el cuerpo de los métodos u otras entidades no referenciadas desde otras clases (*e.g.* propiedades con modificador privado o métodos no exportados).

- **Clientes (*Clients* - C)**: otras clases cliente necesitan ser actualizadas para completar la refactorización. Los cambios afectan a propiedades accesibles utilizadas por clases clientes perteneciendo al mismo o diferente espacio de nombres o espacio de trabajo.

- **Herencia (*Inheritance* - H)**: ancestros o descendientes (participantes en la relación de herencia) son modificados al realizarse la refactorización. La jerarquía es afectada por cambios en propiedades heredadas. El efecto básico es que los cambios deben ser propagados en ambos sentidos (ascendente o descendente) a ancestros y/o descendientes.

Una refactorización puede tener efectos combinados a los tres niveles. Por ejemplo, **Extract Method** [Fowler, 1999], no afecta a otras clases (I), pero **Add Parameter** [Fowler, 1999], aparentemente una refactorización simple, tiene efectos en la clase que contiene el método objetivo (I), en los clientes que usan el método (C), y ancestros o descendientes que definen o redefinen un método (H), por lo que se clasificaría como ICH.

Nivel de conocimiento de diseño y lenguaje de programación *(Design and Language Issues)*

El nivel de conocimientos requerido para aplicar una refactorización apunta algunas pistas sobre su complejidad. Los programadores pueden tener diferente conocimiento y experiencia en el uso del lenguaje de programación, facilitando o dificultando su labor a la hora de implementar refactorizaciones.

Se definen tres niveles de conocimiento, de menor a mayor complejidad:

- **Básico *(Basic)***: una refactorización usa elementos comunes a los LPOO, conceptos manejados por cualquier programador con una experiencia inicial en el uso de un lenguaje de programación en este paradigma.

- **Avanzado *(Advanced)***: conceptos avanzados de POO, como por ejemplo excepciones, aserciones, diseño por contrato, anotaciones, delegados, etc. El uso correcto de estos conceptos requiere una buena base de conocimiento del lenguaje de programación, así como una base sólida de *"buenas prácticas"* en la POO.

- **Patrones de Diseño *(Pattern Design)***: para la aplicación de refactorizaciones, el usuario necesita conocer los fundamentos de los patrones de diseño [Gamma et al., 1995], así como los conceptos de participante, rol, etc. utilizados en su especificación. El diseñador o programador debe estar familiarizado con dichos patrones de diseño y las diferentes soluciones de implementación para cada lenguaje (*idioms*). En relación a la aplicación conjunta de patrones de diseño con las refactorizaciones, la referencia básica es [Kerievsky, 2004].

Algunos ejemplos de clasificación, aplicando estas propiedades son: **Inline Method** (Básico - *Basic*), **Replace Exception with Test** (Avanzado - *Advanced*) y **Form Template Method** (Patrón de Diseño - *Design Pattern*), todas descritas en [Fowler, 1999]. Se asume que cada nivel subsume al siguiente, por lo que aquellos programadores que aplican patrones de diseño, tienen conocimientos avanzados del lenguaje, y lógicamente conocimientos básicos.

Entradas *(Inputs)*

En una definición de una refactorización siempre se necesita especificar las entradas, puesto que son un elemento clave que dirige su ejecución. El usuario (o aplicación si estamos ante una refactorización inferida automáticamente) siempre tiene como objetivo refactorizar a partir de un punto inicial u objetivo, en particular cuando estamos ante refactorizaciones de bajo nivel. Adicionalmente se suministra información, necesaria para completar la ejecución (normalmente cuando la refactorización ofrece diferentes caminos alternativos).

Básicamente el programador necesita identificar:

- **Entrada raíz *(Root input)***: elemento objetivo que el usuario elige como punto de invocación para la ejecución de la refactorización, desde una inspección visual o inferido automáticamente.

  Por ejemplo: **Self Encapsulate Field** [Fowler, 1999], necesita sólo un punto de entrada, el atributo a encapsular. Las acciones a aplicar para llevar a cabo la refactorización no necesitan mayor interacción por parte del usuario.

- **Entradas adicionales *(Additional inputs)***: es muy común que los usuarios provean de información adicional sobre cómo ejecutar la refactorización. Están definidas como entradas adicionales, indicando su número (se optará por un estilo similar a las multiplicidades utilizadas en las asociaciones UML, con valor * para indicar un número indefinido de elementos) y tipo de entrada (correspondiente al tipo de elemento que se maneja en los LPOO) *e.g.* 1 Class * Attributes.

  Los *wizards*, asistentes u otros componentes gráficos están construidos para asistir a los usuarios en la inclusión de estos valores adicionales necesarios para llevar a cabo la refactorización.

  Por ejemplo: **Move Field** [Fowler, 1999] necesita un atributo como entrada raíz, pero el usuario tiene que indicar también la clase destino (entrada adicional) donde se quiere mover el atributo. La interacción con el usuario es obligatoria para elegir la clase destino. El valor de este campo para **Move Field** sería 1 Class.

Dado el contexto del presente trabajo, – *i.e.* programación orientada a objetos– se considerarán las siguientes entradas, ordenadas de mayor a menor nivel de complejidad: clases, parámetros formales, propiedades, métodos, fragmentos de código, instrucciones, expresiones, entidades, atributos, argumentos formales, declaraciones locales y nombres, como elementos

objeto del estudio. El criterio de ordenación utilizado se basa en la relación de uso (o *utilizado en*, *e.g.* un fragmento de código contiene/utiliza instrucciones, una expresión utiliza entidades, etc.), utilizando finalmente el ámbito de la entrada para resolver los empates, como en el caso de las entidades (atributos, argumentos formales y declaraciones locales).

Siempre se necesita una entrada raíz para identificar qué clase de refactorización puede ser aplicada. Este punto es tenido en cuenta en las herramientas actuales. Casi todas ellas deben inferir a partir de la entrada raíz, el conjunto cerrado de refactorizaciones que pueden ser aplicadas a un elemento seleccionado por el usuario. Por lo tanto, necesitamos clasificar las refactorizaciones en función de su elemento raíz, porque permitirá definir el conjunto específico de refactorizaciones que deberían estar habilitadas en una herramienta de refactorización o IDE, cuando se selecciona un elemento en el entorno.

Uno de los problemas actuales que afrontan los usuarios al aplicar refactorizaciones, es la falta de asistencia por parte de las herramientas al acotar el número de refactorizaciones aplicables en función del contexto. Algunos trabajos, como [Emerson Murphy-Hill, 2008] apuntan este problema. Como consecuencia de esto, la comunidad de desarrolladores evita la aplicación de refactorizaciones porque no tienen una idea clara de qué refactorizaciones son aplicables o no en cada momento.

Acciones *(Actions)*

Las acciones se definen como operaciones que cambian el estado del código. Revisando el papel de los *mecanismos* descritos en [Fowler, 1999], o las refactorizaciones de bajo nivel en [Opdyke, 1992], podemos deducir que muchas acciones son llevadas a cabo al ejecutar una refactorización. Estas acciones están vinculadas a la solución de implementación (metamodelos, lógica de predicados, *grammarware*, XML, etc.) pero por motivos de simplicidad sólo un tipo de acción abstracta debería caracterizar la refactorización. Para llevar a cabo dicha abstracción, se consideran las instantáneas del estado previo y posterior a la aplicación de la refactorización, seleccionando la acción principal.

Desde el análisis de los catálogos utilizados (ver Apartado 2.1.4), podemos deducir que la cantidad de acciones candidatas es extensa. Por ejemplo, del catálogo de [Fowler, 1999], se pueden extraer las siguientes acciones: *Extract*, *Inline*, *Replace*, *Introduce*, *Hide*, *Change*, *Encapsulate*, *Separate*, *Pull Up*, *Pull Down*, etc. En el catálogo de [Opdyke, 1992], se pueden extraer: *Create*, *Delete*, *Change*, *Move*, *Composite*, *Make*, *Migrate*, *Add*, etc.

La ampliación de estos catálogos añadiendo nuevas refactorizaciones, provocaría la adición de nuevas acciones. Sin embargo, la caracterización debe ser simple de aplicar y con un conjunto de valores acotado. Por lo tanto, debemos acotar esta propiedad a un conjunto cerrado de acciones, con un pequeño número de opciones, donde cualquier refactorización pueda ser clasificada.

En relación a su similitud con la edición de código fuente, se puede pensar en operaciones básicas sobre texto como copiar, pegar, borrar, encontrar, etc. Con relación al proceso de refactorización, se puede pensar en términos similares, y reducir el número de acciones aplicables, ordenadas de menor a mayor complejidad:

- **Crear** ***(Create)***: construye una nueva instancia de un elemento del modelo.
- **Destruir** ***(Destroy)***: elimina un elemento del sistema.
- **Añadir** ***(Add)***: vincula un elemento al ámbito donde estará contenido.
- **Renombrar** ***(Rename)***: cambia el nombre de un elemento identificado en el sistema.
- **Eliminar** ***(Remove)***: elimina un elemento de un ámbito donde estaba contenido.
- **Reemplazar** ***(Replace)***: como la combinación de operaciones de eliminación y añadido en el mismo ámbito.
- **Mover** ***(Move)***: como la combinación de operaciones de eliminación y añadido en dos instancias diferentes del mismo ámbito.

En este trabajo, no se considerarán las acciones de creación y destrucción, por estar ambas incluidas como acciones intrínsecas en la totalidad de los LPOO. En el primer caso, está contenido en la instanciación de objetos a través de los mecanismos proporcionados por el lenguaje (constructores o métodos de creación), mientras que el segundo caso es una acción realizada generalmente de manera automática en la mayoría de sistemas (recolección de basura) o bien manualmente por el programador a través del concepto de destructores, cuando se elimina algún objeto que no se usa posteriormente. Por lo tanto, no es necesario proporcionar implementaciones adicionales a ambos tipos de acciones, como se verá en el Capítulo 6.

Ejemplos de la caracterización de refactorizaciones según su acción principal serían: **Move** en la refactorización **Move Method**, **Replace** en la refactorización **Replace Constructor with Factory Method** [Fowler, 1999].

Resumen de la caracterización

Resumiendo, el conjunto de categorías y valores asociados para caracterizar las refactorizaciones, son los reflejados a continuación. Se utilizará la terminología en inglés, así como el símbolo | para indica que los valores son exclusivos, y el símbolo & para indicar que pueden combinarse:

- ***Scope***: *Intraclass* (I) & *Client* (C) & *Inheritance* (H).
- ***Knowledge Requirement***: *Basic* | *Advanced* | *Design Pattern.*
- ***Inputs***: clases, parámetros formales, propiedades, métodos, fragmentos de código, instrucciones, expresiones, entidades, atributos, argumentos formales, declaraciones locales y nombres.

  Tal y como se deduce de la aplicación de la caracterización al catálogo en genericidad (ver Sec. 5.5) y al catálogo de Fowler (ver Apéndice B), se detectan los posibles valores para las entradas raíz y adicionales:

  - *Root input*: con valores `Class, FormalPar, Method, CodeFragment, Expression, Entity, Attribute, FormalArgument, LocalDeclaration`.
  - *Additional inputs*: con valores `Class, Property, Method, Instruction, Attribute, Name`.

  Indicando en ambos casos con multiplicidades UML el número mínimo y máximo de ocurrencias.

- ***Action***: *Add* | *Rename* | *Remove* | *Replace* | *Move*.

El requisito básico es que todas estas propiedades puedan ser extraídas de los catálogos desde una observación inicial, sin conocer excesivos detalles del lenguaje de programación objetivo, y la implementación concreta (*idioms*) realizada. Esta solución evita criterios subjetivos, buscando el caso ideal de que diferentes programadores, con distinto bagaje de programación, den iguales valores a las propiedades de una misma refactorización.

La propuesta tiene como debilidad, la no resolución entre los posibles empates a la hora de utilizar la caracterización. Entre aquellas refactorizaciones con iguales valores en todas las propiedades de la caracterización, no se puede resolver nada respecto a su orden, pudiendo darse varias soluciones como válidas. La inclusión de más propiedades podría resolver estos conflictos, pero también dificultaría la utilización de la caracterización rompiendo el requisito inicial de simplicidad en la caracterización a utilizar. Un caso de estudio de su aplicación sobre el catálogo completo de [Fowler, 1999] se recoge en el Apéndice B.

Construyendo interfaces gráficas de usuario

Un problema común con las herramientas de refactorización, es definir una biblioteca gráfica o API que permita construir pantallas que guíen al usuario en su ejecución. Las entradas dan una aproximación de su complejidad, cuanto más grande es el número, más difícil de manejar, con interfaces gráficas mucho más complejas. Ésta es una cuestión básica, relativamente fácil de resolver utilizando las entradas (en número y tipo) como criterio en la clasificación.

Si tenemos refactorizaciones con sólo una entrada raíz (*root input*), puede ser clasificada como automática y podrá ser iniciada automáticamente desde el entorno a partir de una selección del usuario. Por otro lado, si la refactorización necesita información adicional, y por lo tanto necesitamos interacción del usuario, será semiautomática. En el primer caso no se necesita ningún diálogo adicional o asistente, pero con las refactorizaciones semiautomáticas, debe estar disponible alguna clase de soporte gráfico adicional.

Aplicando estos criterios, se pueden clasificar las 68 refactorizaciones [Fowler, 1999] (ver Apéndice B) en dos grandes grupos:

- Automáticas: 31 refactorizaciones *e.g.* **Inline Temp**, **Encapsulate Field**, **Remove Parameter**, **Hide Method**, etc.

- Semiautomáticas: 37 refactorizaciones *e.g.* **Move Method**, **Add Parameter**, **Replace Parameter with Method**, etc.

El siguiente paso es usar la entrada raíz. Dependiendo de dicha entrada, ciertos elementos podrían estar o no activos en el entorno de desarrollo según el elemento seleccionado por el desarrollador. Si además tenemos herramientas que trabajan a nivel de diseño, las refactorizaciones que necesitan un fragmento de código no tendrían sentido y no podrían ser incluidas.

Por otro lado, aplicando la caracterización se pueden agrupar aquellas refactorizaciones que tienen el mismo valor para la propiedad *Root Input*. En función del resultado obtenido, estas refactorizaciones son las únicas que deberían estar disponibles cuando el usuario quiera realizar una refactorización sobre un elemento de dicho tipo.

Desde el punto de vista de reutilización, también necesitamos saber qué refactorizaciones tienen una GUI similar. Estas cuestiones pueden ser respondidas con esta caracterización. Por ejemplo, refactorizaciones como **Extract Class**, **Replace Type Code with Class** y **Replace Record with Data Class**, tienen las mismas entradas: una clase como entrada raíz y * atributos como entradas adicionales. Todas estas refactorizaciones tendrán una interfaz gráfica común, por lo que podrán ser reutilizadas.

Otro caso diferente, es la refactorización **Add Parameter**. Se necesita un método (como entrada raíz) y un argumento formal. No hay otra refactorización en el catálogo que comparta estas entradas, por lo que no podría ser reutilizada su GUI. Si estamos desarrollando una API para interfaces gráficas en refactorización de código, podemos agrupar refactorizaciones con similares entradas para reutilizar un gran número de componentes.

4.3. Proceso de definición de refactorizaciones

La tarea de refactorizar el código es hoy en día una de las piezas básicas de la mayoría de procesos de desarrollo del software, especialmente relevante en metodologías ágiles [Beck, 1999, Beck and Andres, 2004]. Sin embargo, desde un punto de vista del propio desarrollo de las refactorizaciones como producto software, su definición, construcción e integración en herramientas no ha sido abordado desde un punto de vista de proceso.

En esta sección se presenta un proceso que favorece la reutilización. El objetivo es definir un proceso que garantice la construcción de refactorizaciones correctas en cuanto a sus propiedades, en particular en relación a la preservación del comportamiento, pero siempre en un contexto de cierta independencia del lenguaje objetivo del código a refactorizar.

Tomando la estructura previa para la definición de refactorizaciones (ver Sec. 4.1), en [Marticorena et al., 2007a] se razonó sobre los distintos procesos aplicables, con sus ventajas y desventajas, utilizando la notación *Software Process Engineering Metamodel Specificacion (SPEM)* [OMG, 2008]. Como resultado, en el Apartado 4.3.3 se presenta la propuesta de proceso que se seguirá a lo largo del presente trabajo.

4.3.1. Situación previa al estudio

El estudio de las refactorizaciones siempre se enfoca en un lenguaje objetivo concreto más o menos extendido como se ha visto en el Capítulo 2. Sin embargo este enfoque presenta algunos problemas derivados precisamente del hecho de centrarse en único lenguaje objetivo.

Con este enfoque, el proceso es repetido desde cero para incluir refactorizaciones equivalentes sobre otros lenguajes objetivo con características similares. Además en muchos casos las refactorizaciones se implementan desde cero y de forma monolítica sobre el lenguaje objetivo, disminuyendo las posibilidades de reutilización.

Así pues, el proceso y los elementos descritos a continuación toman siempre una doble visión. En primer lugar, se enfoca en aquellos elementos que se pueden generalizar, teniendo en cuenta las propuestas centradas en independencia del lenguaje con el metamodelo MOON. En segundo lugar se enfoca en aquellos elementos particulares de cada lenguaje objetivo.

Los elementos que conforman el proceso, desde el punto de vista de los participantes, las tareas que se realizan en el proceso y los productos resultantes, están orientados a cumplir con esta doble visión.

4.3.2. Elementos del proceso: roles, tareas y productos

Roles

Partiendo de las definiciones previas y la plantilla planteada para la definición de refactorizaciones en el Apartado 4.1.2, se pueden distinguir tres roles básicos en el proceso de refactorización, como se muestra en la Fig. 4.2:

Figura 4.2: Roles en el proceso de definición y construcción de refactorizaciones

Analista de refactorizaciones define los elementos de una refactorización. La definición se basa en sus conocimientos y experiencia sobre el lenguaje objetivo, por lo que se requiere un perfil con altos conocimientos y experiencia en lenguajes de programación. Este rol puede ser asumido habitualmente por el constructor del entorno integrado de desarrollo o herramienta de refactorización. En nuestro caso particular también se deben tener conocimientos en cuanto al metamodelo utilizado (MOON) y de la extensión correspondiente al lenguaje concreto objetivo de las refactorizaciones.

Ensamblador de refactorizaciones busca, construye y ensambla los elementos que componen una refactorización. Los elementos que forman parte de las refactorizaciones pueblan un repositorio al que el ensamblador de refactorizaciones aporta y del que recupera elementos reutilizables. Este rol suele ser asumido por el constructor del entorno integrado de desarrollo o herramienta de refactorización.

Usuario de refactorizaciones utiliza la refactorizaciones, sobre código escrito en un lenguaje de programación concreto, para el que han sido implementadas finalmente las refactorizaciones. De manera adicional cumple un papel de validar la corrección de las refactorizaciones, asistido por herramientas externas como compiladores y entornos de automatización de pruebas, que pueden ayudar a detectar refactorizaciones mal definidas o mal construidas como resultado de su aplicación en sistemas software reales.

Tareas

Habitualmente se supone que el trabajo de ensamblaje y construcción de las refactorizaciones es correcto, llevándose a cabo un proceso de pruebas lo suficientemente completo antes de liberar el producto. Es en la fase posterior de uso/explotación del producto cuando se pueden detectar errores, y la no aplicabilidad de la refactorización debido a una definición incorrecta. Si fuera necesario, los errores recopilados puede ser enviados al analista o ensamblador de refactorizaciones. En siguientes versiones, los errores serán revisados, analizados y corregidos. También se incorporarán nuevas refactorizaciones por parte del analista y ensamblador de refactorizaciones.

Por lo tanto, como tareas básicas en este proceso se pueden distinguir:

Análisis estudio de la refactorización aplicada al lenguaje objetivo concreto sobre el que se trabaja. Es una actividad costosa puesto que exige un estudio detallado de la refactorización aplicada sobre un nuevo lenguaje, con toda la semántica estática y dinámica asociada. Implica también la actividad de revisión de la definición ante posibles fallos posteriores en su aplicación.

Implementación construcción de la parte operativa que es capaz de ejecutar dicha refactorización sobre la representación de código elegida. Obteniendo soluciones más o menos reutilizables, en función de la dependencia concreta sobre el lenguaje objetivo de los elementos implementados que conforman una refactorización.

Refactorización se ejecuta la refactorización sobre casos concretos. Como resultado de la ejecución de la refactorización se puede producir un rechazo preventivo, normalmente aplicado por incumplimiento de precondiciones, o un rechazo posterior a su aplicación, provocado por incumplimiento de postcondiciones, fallo de compilación, error en los tests, etc. Los resultados obtenidos en esta fase pueden retroalimentar las tareas de análisis e implementación.

Productos

En todo proceso es habitual que se utilicen distintos productos. Al refactorizar es necesario aclarar cuál es ese conjunto de productos de entrada y salida sobre los que se trabaja a lo largo del proceso. Como entradas y salidas del proceso se tienen los siguientes productos de trabajo, cuyas asociaciones se muestran en la Fig. 4.3:

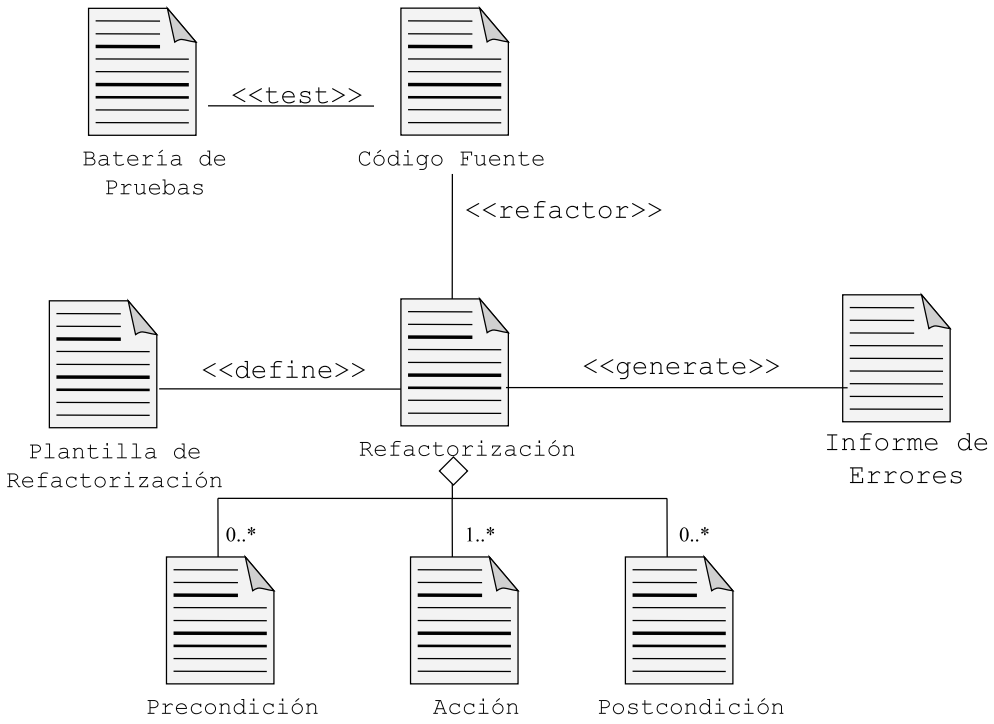

Figura 4.3: Productos en el proceso de definición y construcción de refactorizaciones

Plantilla de refactorización está constituida por los elementos de la definición. Aunque en el presente trabajo se aborda desde el punto de vista de la estructura propuesta (ver Sec. 4.1) puede generalizarse a plantillas de otros autores como [Opdyke, 1992] o la recetas indicadas en [Fowler, 1999].

Refactorización implementación ejecutable de los elementos que constituyen una refactorización. Los elementos se pueden implementar desde cero o bien reutilizarse desde un repositorio ya existente.

Código fuente código a ser transformado, aplicando sobre él la refactorización seleccionada.

Batería de pruebas conjunto de pruebas que comprueban el correcto funcionamiento del código, también entre sucesivas refactorizaciones. Permiten asegurar la preservación del comportamiento, siempre y cuando no sean objeto de las propias refactorizaciones. En

la práctica nos podemos encontrar con sistemas software que no incluyan baterías de pruebas.

Informe de errores registro de errores en la ejecución de una refactorización. Este registro posibilita la acción correctora por parte del analista a la hora de corregir y dar una nueva revisión de la plantilla de refactorización, y por parte del ensamblador a la hora de corregir errores de implementación.

4.3.3. Proceso incremental propuesto para la definición de refactorizaciones

A continuación se propone una forma de abordar el proceso (roles, tareas y productos implicados), enfocada a una mejora de la reutilización y con el objetivo de obtener un mayor grado de independencia del lenguaje, en la Fig. 4.4.

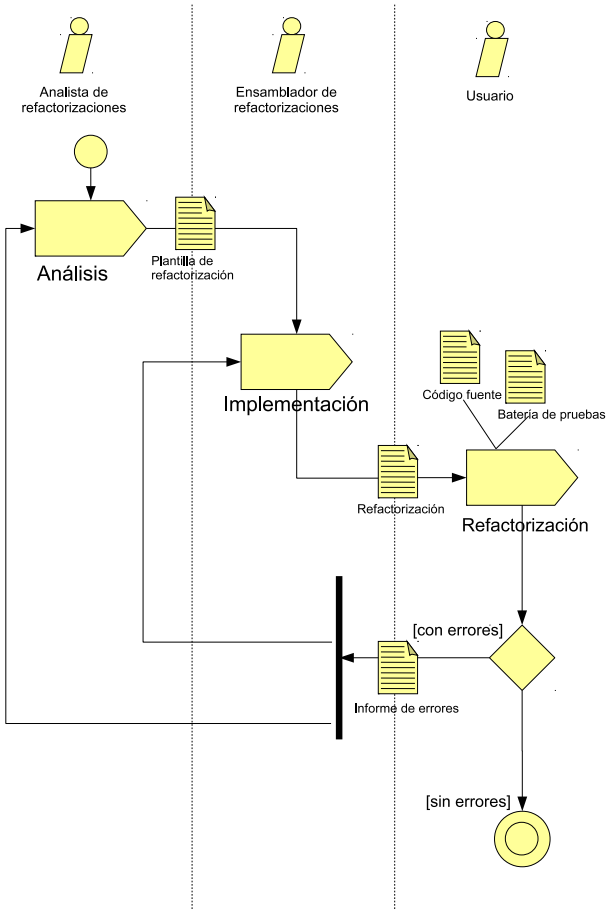

Figura 4.4: Proceso propuesto para la definición, construcción y aplicación de refactorizaciones

Desde el punto de vista de los objetivos de esta tesis, es de especial interés la tarea de análisis, que da como producto final la definición de la refactorización. Dicha definición sigue la estructura previamente establecida en la Sec. 4.1.

Así pues, se propone como proceso para la definición de refactorizaciones un proceso iterativo que incrementalmente complete las definiciones de las refactorizaciones, a medida que se obtienen resultados de una forma exploratoria. En esta solución, se parte de una definición de refactorizaciones, en las que en una primera iteración no se establecen pre ni postcondiciones para el lenguaje objetivo sobre el que se aplica la refactorización. En dicha iteración, se comienza estableciendo precondiciones, acciones y postcondiciones en MOON, y sólo se realiza la particularización de las acciones para el lenguaje concreto. A continuación se va completando la definición en sucesivas iteraciones en el análisis (ver Fig. 4.5) con las pre y postcondiciones particulares del lenguaje concreto.

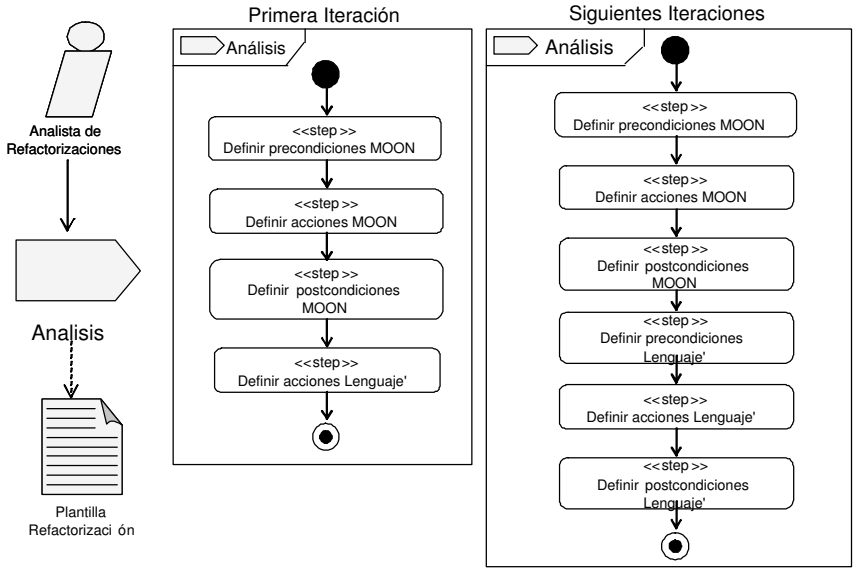

Figura 4.5: Definición incremental de refactorizaciones en el análisis

A medida que se observan rechazos por precondiciones muy restrictivas, o rechazos posteriores por estados incorrectos, se realiza una revisión de los elementos de la refactorización, desde un punto de vista de la independencia del lenguaje y desde un punto de vista del lenguaje particular sobre el que se trabaja. Partiendo de una comprobación relajada, en sucesivas iteraciones se evoluciona a una comprobación estricta.

A lo largo del tiempo el conjunto de pre y postcondiciones particulares del lenguaje, inicialmente vacío en las primeras iteraciones, se completa a partir de la información que se obtiene en sucesivas revisiones (iteraciones). En función de que se disponga de un compilador y el juego de pruebas (en la práctica será habitual disponer de un compilador, pero no siempre de un juego de pruebas), se pueden añadir como pasos opcionales su ejecución, mejorando la preservación de la semántica tanto estática como dinámica del programa refactorizado.

Transcurrido un cierto número de iteraciones se debe llegar a un conjunto adecuado de pre y postcondiciones. Se determinará como condición de parada del proceso una cota de tiempo, sin que la definición de la refactorización haya tenido que ser revisada por errores en su aplicación.

Ventajas

- Desde la primera iteración las precondiciones definidas para MOON permiten rechazar en fase temprana la ejecución de acciones transformadoras costosas.
- En las siguientes iteraciones, las precondiciones definidas respecto a MOON y al lenguaje objetivo, permiten rechazar en fase posteriores la ejecución de acciones transformadoras costosas.
- En las primeras iteraciones, se detectan refactorizaciones incorrectas en relación a la semántica del metamodelo MOON.
- En fase posteriores se detectan refactorizaciones incorrectas en relación a la semántica del lenguaje objetivo.
- Retroalimentación e incremento gradual de los elementos de las refactorizaciones a todos los niveles.

Desventajas

- La estrategia puede ser pesimista, rechazando refactorizaciones cuyo resultado sería correcto respecto al metamodelo.
- En las primeras iteraciones, se pueden llevar a cabo refactorizaciones costosas incorrectas que podrían ser identificadas respecto al lenguaje objetivo.
- Para comprobar la corrección de la refactorización y la preservación del comportamiento se requiere disponer de un juego de pruebas. En la práctica no siempre están disponibles.
- Por motivos de optimización, se debería poder determinar el juego concreto de pruebas a repetir. El coste de determinar dicho juego de pruebas, y el coste de ejecución, aumentan el esfuerzo en la refactorización del código.
- Este enfoque es válido siempre y cuando el juego de pruebas no sufra cambios, motivados por las acciones de transformación en la refactorización. En caso contrario, el propio juego de pruebas pasaría a estar en un estado inconsistente para la demostración de la preservación del comportamiento.

En caso de necesitar aplicar las mismas refactorizaciones o algunos de sus elementos, sobre nuevos lenguajes, se define un proceso que a lo largo de varias iteraciones concluye con una implementación más adecuada. En dicho proceso se reutilizan las definiciones previas,

establecidas con una semántica relajada sólo sobre el metamodelo MOON, pero a su vez permitiendo extenderlas con nuevos elementos propios del lenguaje particular.

En el Capítulo 5 se definirá un catálogo de refactorizaciones en genericidad a partir de la estructura definida en este capítulo, para posteriormente caracterizar dichas refactorizaciones. La solución de soporte a la refactorización, basada en *frameworks*, se detalla en el Capítulo 6, mientras que la aplicabilidad práctica se mostrará en el Capítulo 7, trabajando sobre un lenguaje de programación concreto como es JAVA.

CAPÍTULO 5

CATÁLOGO DE REFACTORIZACIONES SOBRE GENERICIDAD

En el presente capítulo se aborda la construcción de un catálogo de refactorizaciones centradas en genericidad. Las refactorizaciones en dicho catálogo se dividen además desde un doble punto de vista, a través de la generalización y especialización de clases.

Tal y como se indica en [Meyer, 1997], *"para alcanzar metas como la extensibilidad, reutilización y fiabilidad es preciso hacer que la estructura de clase sea más flexible y este esfuerzo está dirigido en dos direcciones."* El autor plantea una dirección vertical, con una visión ascendente de abstracción o generalización y otra descendente o de especialización, y una dirección horizontal para la parametrización de tipos. Dicho esquema es seguido también en el catálogo descrito a continuación, trabajando sobre las clases en ambas direcciones, horizontal y vertical.

Las refactorizaciones del catálogo que se presenta se emparejan en la Tabla 5.1 con su correspondiente refactorización inversa. Entendemos que dos refactorizaciones son inversas cuando una deshace las acciones realizadas por la otra, de forma similar al concepto de función inversa utilizado en matemáticas.

La introducción de la programación genérica en varios de los LPOO fuertemente tipados de la corriente principal como JAVA y C#, o su existencia desde las primeras versiones del lenguaje como en C++ y EIFFEL, refuerza la definición de un conjunto de refactorizaciones centradas en estas características, relativas a las clases y métodos genéricos.

Este catálogo no pretende ser exhaustivo, dada la imposibilidad de predecir el conjunto de refactorizaciones completo, pero sí afronta refactorizaciones enfocadas a la programación genérica que permitan demostrar y validar el uso del metamodelo MOON para la definición de refactorizaciones en este campo. Para la descripción de las refactorizaciones, se ha utilizado

Tabla 5.1: Catálogo de refactorizaciones sobre genericidad

Generalización (Generify) (Sec. 5.3)	**Especialización (Specialize)** (Sec. 5.4)
Parametrizar (Apartado 5.3.1)	Reemplazar parámetro formal por tipo completo (Apartado 5.4.1)
Generalizar acotación (Apartado 5.3.2)	Especializar acotación (Apartado 5.4.2)
Eliminar signaturas en cláusula *tal que* (Apartado 5.3.3)	Añadir signaturas en cláusula *tal que* (Apartado 5.4.3)

el lenguaje ALF [OMG, 2010, OMG, 2012]. En la definición de las refactorizaciones se indican sus motivaciones, entradas, precondiciones, acciones, y postcondiciones si fuera adecuado.

5.1. Descripción de refactorizaciones utilizando el lenguaje ALF

En los catálogos clásicos de refactorizaciones como [Opdyke, 1992, Roberts, 1999b], se razonaba a partir de la información extraída del código con una lógica de predicados. Esta solución es adaptable a una solución basada en metamodelos. Puede tomarse como base el trabajo de [Beckert et al., 2002] donde se planteaba la transformación de diagramas de clases (incluyendo también expresiones OCL) a una lógica de predicados de primer orden con tipos. Con estos trabajos se pueden describir adecuadamente las precondiciones y postcondiciones, pero no la forma de definir las acciones de transformación.

Este punto tampoco se cubría en otros trabajos, como [Tichelaar, 2001]. Toda comprobación se realizaba de manera informal, sin que la correspondencia de las refactorizaciones con su metamodelo (FAMIX) se pueda comprobar de una manera formal, o al menos validar cierto grado de corrección sintáctica y/o semántica. Esta aproximación es similar a la utilizada en GRAMMYUML [Van Gorp et al., 2003a], donde se utilizaba OCL como lenguaje para la definición de pre y postcondiciones, pero de nuevo dejando abierta la definición de acciones que transformen el código y siendo éstas dependientes de la solución particular de implementación dada.

Partiendo de estas experiencias, establecido el metamodelo MOON, la estructura en la definición de refactorizaciones y el lenguaje a utilizar, se debe validar su adecuación a la definición del nuevo catálogo. En concreto para la definición de las consultas sobre la información disponible del código en el metamodelo, así como el soporte de las acciones que modifican el estado de los objetos del mismo. El problema posterior de la recuperación del código refactorizado se abordará en el Capítulo 6.

Tal y como se introdujo en Capítulo 4, se ha escogido el lenguaje ALF *(Action Language for Foundational* UML*)* [OMG, 2010, OMG, 2012] como lenguaje formal para la especifica-

ción de las refactorizaciones. ALF es una representación textual de elementos de modelado UML. La semántica de ejecución se establece por una correspondencia de su sintaxis concreta a la sintaxis abstracta del estándar *Foundational* UML (FUML) [OMG, 2011]. Así pues, el resultado de la ejecución de un fichero de entrada ALF está dado por la semántica del modelo FUML, definido en su especificación.

Frente a las limitaciones que impone un lenguaje de restricciones como OCL, el lenguaje ALF proporciona un conjunto de características adecuado, mucho más amplio. Sin pretender dar una descripción exhaustiva, el lenguaje ALF se adecúa a la definición de refactorizaciones sobre un metamodelo, brindando el siguiente conjunto de oportunidades:

- Permite la representación estructural completa de un modelo (*e.g.* clases, asociaciones, multiplicidades, restricciones, etc.)
- Permite la navegación sobre los objetos del modelo, sus asociaciones y el acceso/manipulación de las instancias.
- Permite la especificación de comportamientos ejecutables en dichos modelos representados con UML (*e.g.* actividades, como elemento fundamental en ALF).
- Establece un sistema implícito de tipos (no requiere declaración explícita de tipos en una actividad) pero siempre proporciona comprobación de tipos en las operaciones realizadas (*e.g.* asignaciones, paso de mensaje, etc.)
- Incluye expresiones, y más concretamente expresiones de expansión sobre secuencias de elementos permitiendo cuantificadores (*e.g.* `forAll`, `exists`) y consultas (*e.g.* `select`). Esto permite definir consultas de forma declarativa sobre el modelo con equivalencia a una lógica de predicados de primer orden.
- Incluye el concepto de sentencia y bloque de sentencias, junto a un conjunto amplio de sentencias predefinidas equivalente a instrucciones de control de flujo (*e.g.* `if`, `for`, `switch`), permitiendo también una definición imperativa en las actividades.
- Define un conjunto básico de tipos (*e.g.* `Boolean`, `String`, `Integer`, etc.) y bibliotecas predefinidas de estructuras de datos genéricas (*e.g.* `Collection`, `List`, `Set`, etc.)
- Permite realizar la comprobación de la definición de una refactorización. Se puede comprobar la corrección de la sintaxis y la semántica estática con las herramientas asociadas a ALF.

Por otro lado, se pueden apuntar algunas amenazas en cuanto a su uso:

- Falta de madurez en el desarrollo de herramientas que faciliten la transición de metamodelos como MOON a su correspondiente definición en ALF, aunque existen trabajos en curso [CEA, 2013].

- Actualmente, no existen herramientas para generar código ejecutable a partir de ALF. El compilador utilizado [Data Access Technologies Inc, 2012] realiza una comprobación sintáctica y de semántica estática, no genera código, pese a que este es uno de los objetivos finales del lenguaje de acciones. Esto impide poder realizar una comprobación de la preservación de la semántica dinámica.

- El hecho de que ALF proporcione una sintaxis tanto declarativa como imperativa, puede conducir a expresar las actividades en una forma condicionada a su implementación posterior en un lenguaje particular.

- La especificación de ALF está en una versión Beta en OMG, pendiente todavía de modificaciones. Aunque existen herramientas que ya están integrando ALF, pudiera estar todavía sometida a cambios.

Pese a las amenazas descritas, el conjunto de oportunidades que brinda, a la hora de la definición de refactorizaciones, apuntan a su adecuación para lograr los objetivos fijados en el presente trabajo.

En la Fig. 5.1 se muestra el flujo de trabajo para el uso de ALF que se propone para la definición y construcción de refactorizaciones. Partiendo del metamodelo MOON, se define en ficheros de texto con sintaxis ALF (extensión `.alf` por convención) tanto el metamodelo, como las actividades que realizan consultas (funciones y predicados) y que modifican el estado de los objetos (acciones). La validación se realiza a través de un *parser* implementado en JAVA [Data Access Technologies Inc, 2012] con el que se comprueba la corrección y la preservación de restricciones sintácticas y de semántica estática del lenguaje.

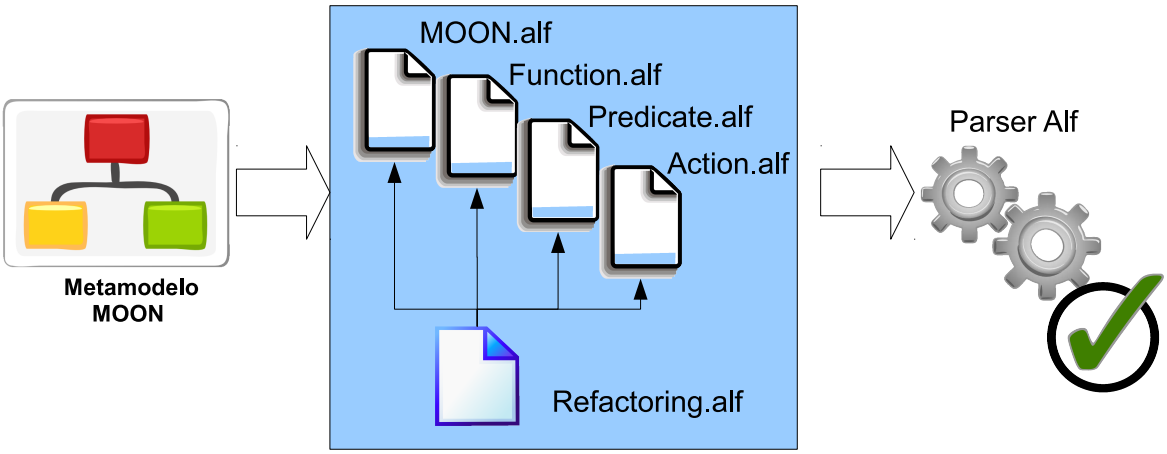

Figura 5.1: Flujo de trabajo usando ALF en la definición de refactorizaciones

Las siguientes secciones muestran cómo, a partir de la definición del metamodelo MOON presentado en el Capítulo 3, se establecerán el conjunto de funciones, predicados y acciones,

utilizando ALF como lenguaje de descripción. A partir de estos elementos se definirá el catálogo de refactorizaciones sobre genericidad, detallado posteriormente en las Sec. 5.3 y Sec. 5.4.

5.2. Funciones básicas

Las funciones básicas permiten consultar el estado actual del código. Son utilizadas tanto en la construcción de predicados, tomando el rol de pre o postcondiciones, como en la construcción de acciones que modifican el estado del código.

En trabajos previos se planteó dicha definición a través de una lógica de predicados de primer orden con tipos, utilizando como base la gramática de MOON [Marticorena et al., 2003, López et al., 2006b, Marticorena and Crespo, 2008, Marticorena et al., 2010a, Marticorena et al., 2011b]. A partir de lo expuesto en la Sec. 5.1, dadas las ventajas aportadas frente a soluciones basadas en lógica de predicados, se asume ALF como base para definir las refactorizaciones, y por tanto también las funciones básicas, reflejando los primeros resultados en recientes trabajos como [Marticorena and Crespo, 2012].

A continuación se presenta la descripción textual del conjunto de funciones que permiten consultar aspectos relativos a la genericidad, mientras que en el Apéndice C y Apéndice D, se incluyen las especificaciones detalladas en ALF de las funciones aquí descritas.

5.2.1. `DependantDownFormalPar`

Descripción: dado un parámetro formal de una clase, devuelve todos los parámetros formales dependientes hacia abajo en clases **descendientes** y **clientes**. Un parámetro formal es dependiente hacia abajo de otro cuando:

- Un parámetro formal en el descendiente ocupa la misma posición de un parámetro formal del ancestro en la cláusula de herencia, pasando éste a ser dependiente. Esto siempre depende de la correspondencia en la posición ordenada en la lista de parámetros formales en la declaración de la clase ancestro, y en el uso de la misma en la cláusula de herencia del descendiente. Es decir la correspondencia no es por nombre, de la misma forma que se produce la correspondencia por posición entre argumentos formales y argumentos actuales en la invocación a métodos.

  Por ejemplo, dadas las siguientes declaraciones de clases descritas utilizando el lenguaje minimal MOON:

```
class Ancestor[T,S] body end;

class Descendant1[U] inherit Ancestor[U, Object]
body end;

class Descendant2[V] inherit Ancestor[Object, V]
body end;
```

El parámetro formal `U` en la clase `Descendant1` es dependiente de `T` en la clase `Ancestor`. Por otro lado, el parámetro formal `V` en la clase `Descendant2` es dependiente de `S` en la clase `Ancestor`.

- Un parámetro formal en una clase cliente, se corresponde con el parámetro formal que es utilizado como sustitución (parámetro actual) del correspondiente parámetro formal de la clase genérica proveedora. De nuevo, depende de la posición ordenada en la lista de parámetros formales declarados en el proveedor y sustituidos en la clase cliente, sin requerir coincidencia de nombre.

  Por ejemplo, dadas las siguientes declaraciones de clases en MOON:

```
class Provider[T,S] body end;

class Client1[U]
signatures
    attributes
        v1 : Provider[U, Object];
body
end;

class Client2[V]
signatures
    attributes
        v2 : Provider[Object, V];
body
end;
```

  El parámetro formal `U` en la clase `Client1` es dependiente de `T` en la clase `Provider` a través de la declaración de la variable `v1`. Por otro lado, el parámetro formal `V` en la clase `Client2` es dependiente de `S` en la clase `Provider` a través de la declaración de la variable `v2`.

El proceso de búsqueda se repite recursivamente para los descendientes y clientes de clases en los que se ha detectado un parámetro formal dependiente hacia abajo, hasta que no quedan nuevos parámetros formales por visitar.

Esta función es simétrica a la función `DependantUpFormalPar`, pero en el sentido descendente respecto a la jerarquía de herencia, y de las relaciones de cliente entre las clases.

5.2.2. `DependantUpFormalPar`

Descripción: dado un parámetro formal de una clase, devuelve todos los parámetros formales dependientes hacia arriba en clases **ancestros** y **proveedores**. Un parámetro formal es dependiente hacia arriba de otro cuando:

- Un parámetro formal en el ancestro es sustituido en la cláusula de herencia del descendiente por un parámetro formal. Así el parámetro formal del ancestro pasa a ser dependiente hacia arriba del parámetro del descendiente. Esto siempre depende de la posición ordenada en la lista de parámetros formales en la declaración en el ancestro

y en el uso en el descendiente, aunque no exista coincidencia de nombre, de la misma forma que se produce la correspondencia entre argumentos formales y argumentos actuales en la invocación a métodos.

Por ejemplo, dadas las siguientes declaraciones de clases descritas utilizando el lenguaje minimal MOON:

```
class Ancestor[T,S] body end;

class Descendant1[U] inherit Ancestor[U, Object]
body end;

class Descendant2[V] inherit Ancestor[Object, V]
body end;
```

El parámetro formal `T` en la clase `Ancestor` es dependiente hacia arriba de `U` en la clase `Descendant1`. Por otro lado, el parámetro formal `S` en la clase `Ancestor` es dependiente hacia arriba de `V` en la clase `Descendant2`.

- Un parámetro formal en una clase proveedora es dependiente hacia arriba del parámetro formal que es utilizado como sustitución (parámetro actual) del correspondiente parámetro formal en la clase genérica cliente. De nuevo, depende de la posición ordenada en la lista de parámetros formales declarados en el proveedor y sustituidos en el cliente, sin requerir coincidencia de nombre.

  Por ejemplo, dadas las siguientes declaraciones de clases:

```
class Provider[T,S] body end;

class Client1[U]
signatures
    attributes
        v1 : Provider[U, Object];
body
end;

class Client2[V]
signatures
    attributes
        v1 : Provider[Object, V];
body
end;
```

  El parámetro formal `T` en la clase `Provider` es dependiente hacia arriba de `U`, parámetro formal de la clase `Client1`, a través de la declaración de la variable `v1`. Por otro lado, el parámetro formal `S` en la clase `Provider` es dependiente hacia arriba de `V` de la clase `Client2`, a través de la declaración de la variable `v2`.

El proceso de búsqueda se repite recursivamente para los ancestros y proveedores de clases en los que se ha detectado un parámetro formal dependiente hacia arriba, hasta que no quedan nuevos parámetros formales por visitar.

Esta función es simétrica a la función `DependantDownFormalPar`, pero en el sentido ascendente respecto a la jerarquía de herencia, y de las relaciones de proveedor entre las clases.

5.2.3. RealSubstitutionsOfFormalPar

Descripción: dado un parámetro formal de una clase, devuelve todas las sustituciones por tipos definidos por clases (`ClassType`) a dicho parámetro formal.

Por ejemplo, dadas las siguientes declaraciones de clases descritas utilizando el lenguaje minimal MOON:

```
class Provider[T] body end;

class A
body end;

class Client1[U]
signatures
    attributes
        v1 : Provider[A]
body end;

class Client2[V]
signatures
    attributes
        v2 : Provider[Client1[V]]
body end;
```

Como sustituciones reales del parámetro formal `T` en la clase `Provider`, se tiene al tipo completamente instanciado `A` y al tipo paramétrico no completamente instanciado `Client1[V]`.

5.2.4. CollectSignaturesToAdd

Descripción: dado un parámetro formal de una clase y a partir del conjunto de tipos definidos por clase que se utilizan como sustitución del parámetro formal obtenidos con la función `RealSubstitutionsOfFormalPar` y de los parámetros formales dependientes hacia abajo calculados con la función `DependantDownFormalPar`, calcula el conjunto de métodos que tienen todos estos tipos en común a través de la función `CollectMethodIntersection`.

5.2.5. CollectMethodIntersection

Descripción: dado un conjunto de tipos definidos por clase, calcula la intersección del conjunto de métodos que tienen igual signatura y pertenecen a todas las clases determinantes de dichos tipos.

5.2.6. CollectEntitiesWithType

Descripción: dado un parámetro formal de una clase, devuelve el conjunto de entidades de la clase cuyo tipo es igual al parámetro genérico formal.

5.2.7. CollectMethodsUsedByEntity

Descripción: dada una entidad, devuelve el conjunto de métodos invocados a través de dicha entidad.

5.2.8. CollectDownRealSubstitutions

Descripción: dado un parámetro formal devuelve todas las sustituciones por tipos definidos por clases en parámetros formales dependientes hacia abajo.

5.2.9. CollectPropertiesUsedByEntity

Descripción: dada una entidad, devuelve el conjunto de las propiedades invocadas por dicha entidad en el conjunto de instrucciones.

5.2.10. DependantDescendantsProvidersBoundW

Descripción: dado un parámetro formal de una clase, con acotación por cláusulas *tal que*, devuelve todos los parámetros formales dependientes hacia abajo en clases descendientes, así como los parámetros formales dependientes en proveedores de dichas clases, siempre y cuando contengan el método en su acotación.

Depende de dos funciones auxiliares: `CollectDependantProvidersBoundW` y `CollectDependantDescendantsBoundW`. En contraposición a otras funciones como `DependantDownFormalPar` y `DependantUpFormalPar` donde se especifican utilizando una única función auxiliar, en este caso el tratamiento entre proveedores y descendientes varía ligeramente, obligando a utilizar dos funciones auxiliares.

La actividad `CollectDependantProvidersBoundW`, repite el proceso de búsqueda para los tipos dependientes en proveedores, siempre y cuando contengan el método indicado en la acotación. En caso de no contener el método no es necesario proseguir el análisis.

La actividad `CollectDependantDescendantsBoundW`, repite el proceso de búsqueda para los tipos dependientes, en los descendientes.

5.2.11. ParameterizeUniverse

Descripción: dada una entidad, se define como conjunto de clases condicionantes de la clases objetivo (contiene a la entidad de entrada), y por la consideración especial de las clases descendientes, todas las clases descendientes de la clase que contiene a dicha entidad [Crespo, 2000]. El conjunto se recopila recursivamente, tomando como punto de inicio la clase con la entidad de entrada y finalizando cuando en una iteración no se ha modificado el conjunto.

5.2.12. `EquivalentEntities`

Descripción: dada una entidad se define como el conjunto de propiedades equivalentes en los caminos de herencia a dicha propiedad en otras clases. Mientras que en el caso de las entidades internas (*e.g.* variables locales) no hay equivalencia, en el caso de los atributos, resultados de funciones y argumentos formales, pueden existir varias entidades equivalente como resultado de la redeclaración de las mismas en otras clases de la jerarquía de herencia.

5.2.13. `GDD`

Descripción: dada una entidad, se define como el conjunto de entidades cuyo tipo cambia, si el tipo de la entidad de entrada cambiara para ser un parámetro formal.

5.2.14. `GDI`

Descripción: dada una entidad, se define como el conjunto de entidades cuyo tipo, al cambiar el tipo de la entidad de entrada para ser un parámetro formal, cambia a un nuevo tipo paramétrico donde su parámetro actual es un parámetro formal.

5.2.15. `GenericDependant`

Descripción: dada una entidad, se define como la unión de los conjuntos de entidades genérico dependientes directas (GDD) o genérico dependientes indirectas (GDI) de dicha entidad. Se establece como un algoritmo de punto fijo, en el que se van añadiendo entidades, hasta que se llega a una iteración en la que el conjunto no varía.

5.2.16. `ExpressionsAssignedToEntity`

Descripción: dada una entidad, consulta el conjunto de todas las expresiones asignadas a dicha entidad en el código.

5.2.17. `CallExprUsedByEntity`

Descripción: dada una entidad, devuelve el conjunto de llamadas a expresiones realizadas a través de una determinada entidad.

5.2.18. `CallExprUsingEntity`

Descripción: dada una entidad, devuelve el conjunto de llamadas a expresiones que utilizan a una determinada entidad.

5.3. Catálogo en generalización

Como se indicó al principio de este capítulo, la mejora de la calidad del software que proporciona la genericidad [Meyer, 1997], en particular en cuestiones muy próximas al campo de la refactorización, como son la facilidad de mantenimiento y nivel de reutilización, es un motivo fundamental para definir un catálogo en generalización.

La generalización de los parámetros genéricos aumenta el grado de reutilización de las clases genéricas, ampliando el conjunto de posibles sustituciones válidas. Sin embargo, este aumento de posibilidades de reutilización, disminuye el posible conjunto de propiedades disponibles en las entidades cuyo tipo viene dado por el parámetro genérico formal. Como se ve en la Sec. 5.4, se pueden definir las refactorizaciones inversas, en el sentido de la especialización, cuando se quiere refactorizar el código con el objetivo contrario.

A continuación se detalla el conjunto de refactorizaciones ligadas a la inclusión de la genericidad y la generalización respecto a las posibles sustituciones de los parámetros genéricos.

5.3.1. Refactorización Parametrizar *(Parameterize)*

En algunas situaciones, se descubre la necesidad de parametrizar tipos posteriormente a la construcción de la clase que lo implementa. Bien para trabajar con diferentes tipos con una única definición de clases, o bien para migrar código entre diferentes versiones del lenguaje para adaptarse al uso de genericidad.

Los entornos deben aportar algún tipo de soporte a la transformación automática de antiguas a nuevas soluciones, con mínima interacción por parte del usuario. Para cumplir estos requisitos, es necesario proveer alguna refactorización que permita cambiar de clases no genéricas a genéricas.

Esta refactorización se corresponde *parcialmente* con la operación **Parameterize** [Crespo, 2000]. La refactorización tal y como se plantea en dicho trabajo, se puede encuadrar dentro de la clasificación de [Fowler, 1999] en el grupo de *BigRefactoring*, o bien en la clasificación de [Opdyke, 1992] en el grupo de *High Level Refactoring*. Por lo tanto estamos ante una refactorización de alto nivel, que como bien se indica en diferentes puntos [Crespo, 2000], puede requerir intervención por parte del programador a la hora de aplicarla. Como resultado de la aplicación de dicha refactorización, se espera la generalización de la clase objetivo y sus condicionantes en el repositorio, acotando los parámetros genéricos:

- En las variantes de acotación por subtipado/conformidad, por el tipo previo de la entidad de la parametrización.
- En la variante de acotación por cláusulas tal que (*where*) añadiendo en la acotación las propiedades del tipo previo de la entidad de la parametrización.

En la solución presentada en este trabajo, se presenta la solución para la variante de acotación por subtipado o conformidad. Consecuentemente, la refactorización correspondiente en

el catálogo –desde el punto de vista de generalización y especialización (**Reemplazar parámetro formal por tipo completo**, ver Apartado 5.4.1)– también se ha abordado para las variantes de acotación, subtipado o conformidad.

Descripción

Reemplazar un tipo completo de una entidad por un parámetro formal. El parámetro formal tiene como nueva acotación al tipo de entrada de la entidad sustituido (ver Fig. 5.2).

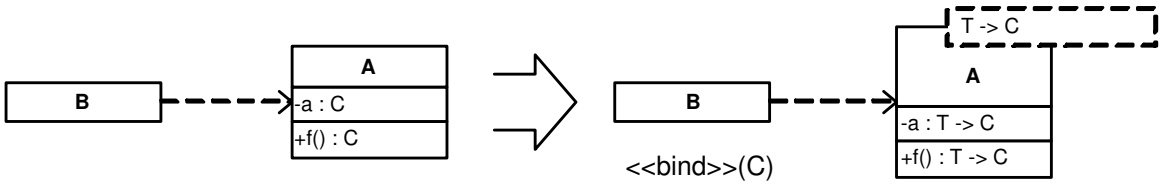

Figura 5.2: Parametrizar: Reemplazar tipo completo por un parámetro formal

Motivación

En una versión inicial de la clase no se consideró el uso de genericidad. El motivo pudo surgir de no considerar la necesidad de la variación de tipos en el sentido horizontal, o la simple ausencia de dicha característica en el lenguaje de programación objetivo con el que se implementó la solución actual. A posteriori, se detecta la necesidad, considerando además que el lenguaje de programación objetivo soporte dicha característica.

Análisis

Para cada entidad dependiente en tipo del parámetro formal, entidades genérico dependientes directos y entidades genérico dependientes indirectas (denominados GDD y GDI respectivamente en [Crespo, 2000]) se reemplaza el tipo por el nuevo parámetro formal, o bien se define el tipo paramétrico en función del nuevo parámetro formal respectivamente.

En las clases **clientes**, incluyendo clases genérico dependientes indirectas, se añade tanto en la declaración de tipo, como en todas las instanciaciones genéricas, el tipo previo de la entidad de entrada como parámetro actual.

En los **ancestros** no se realizan cambios mientras que en los **descendientes** se cambia en la cláusula de herencia el tipo previo por el nuevo tipo paramétrico con la sustitución del parámetro formal por el tipo completo de la entidad de entrada previo a la refactorización.

Entradas

- Entidad `entity : Entity`.
- Nombre del nuevo parámetro formal `name : Name`.

Precondiciones

Precondición 1: Si la clase actual a la que pertenece la entidad es genérica, el nombre del nuevo parámetro formal que se propone en la invocación del operador, tiene que ser distinto a cualquiera de los identificadores de los parámetros genéricos formales de las clases objetivo de la refactorización. Se utiliza el predicado `NotExistsFormalParWithEqualName`.

Código 5.3.1: Predicado `NotExistsFormalParWithEqualName`

```
namespace Moon::Predicate;

public import Moon::Function::ParameterizeUniverse;

activity NotExistsFormalParWithEqualName(in classDef : ClassDef, in  name : Name) :
    Boolean
{
        if (classDef.isGeneric()) {
                List<ClassDef> classes = ParameterizeUniverse(classDef);
                return ! (classes.toSequence() ->
            exists aux ((aux.isGeneric()) &&
                        (aux.getFormalPars().toSequence() -> exists fp (fp.getName() ==
                             name))));
        } // if
        return true;
}
```

Precondición 2: No se permite transformar una clase genérica para cambiar el tipo de una entidad cuyo tipo ya esté expresado en función de uno de los parámetros formales de la clase. Se utiliza el predicado `TargetEntityWithNotVarType`.

Código 5.3.2: Predicado `TargetEntityWithNotVarType`

```
namespace Moon::Predicate;

activity TargetEntityWithNotVarType(in entity : Entity) : Boolean
{
        return entity.getType().isComplete();
}
```

Precondición 3: Si la entidad de la refactorización o cualquier entidad equivalente, introduce una recursividad en su clase, entonces dicha clase no puede ser parametrizada. Como se describe en [Crespo, 2000], un tipo recursivo es aquel cuya especificación está definida en términos del propio tipo.

Dependiendo de la posición, en las signaturas de las propiedades, que tiene la entidad cuya anotación de tipo introduce la recursividad, se define en [Canning et al., 1989] que la entidad introduce una recursividad positiva o negativa. Se utiliza el predicado `TargetEntityNotRecursive`, que se define en función del predicado `IsRecursivePositive` (ver Código 5.3.4) y del predicado `IsRecursiveNegative` (ver Código 5.3.5).

Código 5.3.3: Predicado `TargetEntityNotRecursive`

```
namespace Moon::Predicate;

public import Moon::Function::EquivalentEntities;

activity TargetEntityNotRecursive(in entity : Entity) : Boolean
{
        List<Entity> entities = EquivalentEntities(entity);
    return entities.toSequence() ->
                forAll auxEntity (!IsRecursivePositive(auxEntity) && !
                    IsRecursiveNegative(auxEntity));
}
```

Una entidad introduce una recursividad positiva si la entidad introduce una recursividad y es un atributo o el resultado de una función. A esta entidad se le llama entidad recursiva positiva.

Código 5.3.4: Predicado `IsRecursivePositive`

```
namespace Moon::Predicate;

activity IsRecursivePositive(in entity : Entity) : Boolean
{
        if (entity.getType() instanceof ClassType &&
          (entity instanceof Attribute || entity instanceof Result)){
                ClassDef cd = ((ClassType) entity.getType()).getClassDef();
                return (cd.getClassType() == entity.getType());
        }
        return false;
}
```

Por el contrario, se dice que una entidad introduce una recursividad negativa si la entidad introduce una recursividad y es un argumento de un método. A esta entidad se le llama entidad recursiva negativa.

Código 5.3.5: Predicado `IsRecursiveNegative`

```
namespace Moon::Predicate;

public import Moon::Function::Entities;

activity IsRecursiveNegative(in entity : Entity) : Boolean
{
  if (entity instanceof FormalArgument){
        Method methDec = ((FormalArgument) entity).getMethDec();
```

```
        ClassDef cd = methDec.getClassDef();
        return (cd.getClassType() == entity.getType());
  }
  return false;
}
```

Precondición 4: La intersección entre `GDD` y `GDI` de la entidad de entrada debe ser vacía. Se utiliza el predicado `NotExistsInBothSets`.

Código 5.3.6: Predicado `NotExistsInBothSets`

```
namespace Moon::Predicate;

public import Moon::Function::GenericDependant;

activity NotExistsInBothSets(in entity : Entity) : Boolean
{
        List<Entity> gdd = new List<Entity>();
        List<Entity> gdi = new List<Entity>();
        gdd = GenericDependant(entity, gdi);
        return gdi -> forAll entityGdd (! (gdd -> exists entityGdi(entityGdi ==
            entityGdd)));
}
```

Precondición 5: Si en una clase del conjunto objetivo de la refactorización se encuentra una asociación a una entidad genérico dependiente directa donde a la entidad se le asocia una constante manifiesta distinta de nulo, entonces no puede ser parametrizada. La clase no es lo suficientemente general para ser una clase genérica respecto a esa entidad, al utilizar constantes manifiestas, se compromete con un tipo de entidades. Se utiliza el predicado `NotAssignmentWithConstant`.

Código 5.3.7: Predicado `NotAssignmentWithConstant`

```
namespace Moon::Predicate;

public import Moon::Function::GenericDependant;
public import Moon::Function::ExpressionsAssignedToEntity;

activity NotAssignmentWithConstant(in entity : Entity) : Boolean
{
        List<Entity> gdi = new List<Entity>();
        List<Entity> gdd = new List<Entity>();
        gdd = GenericDependant(entity,gdi);
        return gdd.toSequence() ->
            forAll entityGdd (ExpressionsAssignedToEntity(entityGdd).toSequence() ->
                forAll expr (!expr.isConstant() || expr instanceof Nil));
}
```

Acciones

Acción 1: se realizan los cambios propios de la parametrización del tipo, añadiendo el parámetro genérico a la clase, cambiando de tipo todas las entidades dependientes en tipo,

tanto directa o indirectamente, así como las declaraciones de tipos en los clientes de las clases. Se define la acción `ParameterizeAction`.

Código 5.3.8: Acción `ParameterizeAction`

```
namespace Moon::Action;

public import Moon::Function::GenericDependant;

activity ParameterizeAction(in entity : Entity, in name : Name) {
        // Changed classes
        List<ClassDef> historyClasses = new List<ClassDef>();
        // Changed entities
        List<Entity> historyEntity = new List<Entity>();
        // Candidate classes from GDI entities
        List<Entity> gdi = new List<Entity>();
        List<Entity> gdd = GenericDependant(entity, gdi);
        List<ClassDef> candidates = new List<ClassDef>(gdi.toSequence() -> collect
            entityGdi (entityGdi.getClassDef()));
        // Save previous type of the target entity
        Type oldTargetEntityType = entity.getType();

        // Step 1: add formal parameter to the class
         AddFormalParameterInCandidateClassesAction(candidates, name,
             oldTargetEntityType);
        // Step 2: change type of generic dependant direct entities
        ChangeGDDEntitiesAction(entity, name);
        // Step 3: change type of generic dependant indirect entities
        ChangeGDIEntitiesAction(historyClasses, historyEntity, candidates,
            oldTargetEntityType);
        // Step 4: change type declaration of GDI clients
        ChangeClientsOfGDIEntitiesAction(historyEntity, candidates,
            oldTargetEntityType);
}
```

Por motivos de brevedad, no se detalla el código ALF correspondiente a las cuatro acciones auxiliares en dicha refactorización, estando disponible una definición detallada en el Apéndice D.

Acción 2: se eliminan los moldeados (cast) no necesarios por el uso de entidades genérico dependientes de la entidad de la refactorización. En las asignaciones con entidades que pasan a tener un tipo definido por el parámetro genérico introducido, se elimina la utilización de los moldeados. La comprobación de la corrección en cuanto a tipos pasa a ser labor del compilador, utilizando la información explícita en la declaración de tipos de las variables. Se define la acción `RemoveCastAction`.

Código 5.3.9: Acción `RemoveCastAction`

```
namespace Moon::Action;

public import Moon::Function::GDD;
public import Moon::Function::GDI;
public import Moon::Function::CallExprUsingEntity;

activity RemoveCastAction(in entity : Entity){
```

```
        // Step 1: Remove cast from GDD
        List<Entity> gddListEntity = GDD(entity);
        for (Entity gddEntity : gddListEntity) {
                // Remove cast from result entity
                RemoveCastFromResultAction(gddEntity);
                List<CallExpr> listCallExpr = CallExprUsingEntity(entity);
                for(CallExrp callExpr : listCallExpr) {
                        if (callExpr.hasCastType()) {
                                callExpr.removeCastType();
                        }
                }
        }

        // Step 2: Remove cast from GDI
        List<Entity> gdiListEntity = GDI(entity);
        for (Entity gdiEntity : gdiListEntity) {
                RemoveCastFromResultAction(gdiEntity);
        }
}
```

Esta acción reutiliza a su vez la acción `RemoveCastFromResultAction`. Dicha acción elimina el moldeado en los resultados de las funciones que pasan a tener su retorno definido por el parámetro genérico formal introducido en la refactorización.

Código 5.3.10: Acción `RemoveCastFromResultAction`

```
namespace Moon::Action;

public import Moon::Function::CallExprUsedByEntity;

activity RemoveCastFromResultAction(in entity : Entity){
        List<CallExpr> callExprList = CallExprUsedByEntity(entity);
        for (CallExpr callExpr : callExprList) {
                if (callExpr.getTarget() instanceof Result
                                && callExpr.getCastType() != null) {
                        Result result = (Result) callExpr.getTarget();
                        FunctionDec function = (FunctionDec) result.getFunctionDec();
                        if (function.getReturnType() instanceof FormalPar) {
                                // remove cast
                                callExpr.removeCastType();
                        }
                }
        }
}
```

Restricciones del lenguaje de programación

Aplicable con lenguajes de programación que incluyan programación genérica con acotación por subtipado/conformidad como Java, C# o Eiffel.

Refactorización

La refactorización **Parametrizar** queda finalmente construida de la siguiente forma expresada en Alf:

Código 5.3.11: Refactorización **Parametrizar**

```
namespace Moon::Refactoring;

public import Moon::Predicate::NotExistsFormalParWithEqualName;
public import Moon::Predicate::TargetEntityWithNotVarType;
public import Moon::Predicate::TargetEntityNotRecursive;
public import Moon::Predicate::NotExistsInBothSets;
public import Moon::Predicate::NotAssignmentWithConstant;
public import Moon::Action::ParameterizeAction;
public import Moon::Action::RemoveCastAction;

activity ParameterizeRefactoring(in entity : Entity, in name : Name) : Boolean
{
        // Pre:
        if (NotExistsFormalParWithEqualName(entity.getClassDef(), name) &&
                TargetEntityWithNotVarType(entity) &&
                TargetEntityNotRecursive(entity) &&
                NotExistsInBothSets(entity) &&
                NotAssignmentWithConstant(entity))
        {
                // Action:
                ParameterizeAction(entity, name);
                RemoveCastAction(entity);
        }
        // Post:
        return true;
}
```

5.3.2. Refactorización Generalizar acotación *(Generalize bound)*

En algunas ocasiones, la acotación sobre el parámetro formal de la clase es demasiado estricta. Esto se puede observar a partir del análisis de todas las entidades dependientes en tipo de dicho parámetro.

Si el conjunto de propiedades utilizado sobre todas ellas, es un subconjunto de las propiedades del tipo acotación, y existe un supertipo, cuyo conjunto de propiedades coincida con el conjunto calculado, la acotación puede relajarse.

Descripción

Reemplazar la acotación de un parámetro formal por un supertipo de la misma (ver Fig. 5.3).

Motivación

En una versión inicial de la clase se planteó una acotación mucho más estricta de lo necesario. En la práctica todas las entidades con tipo definido por el parámetro formal, utilizan propiedades definidas en alguno de los ancestros de la acotación del parámetro formal, y por lo tanto se puede relajar la acotación.

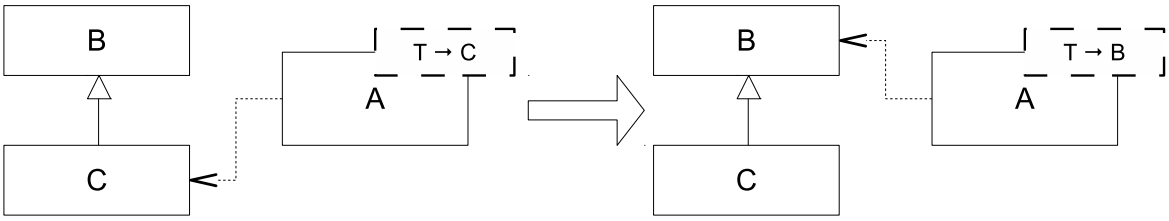

Figura 5.3: Generalizar acotación

Análisis

Dada una clase con acotación por subtipado y una serie de clases dependientes –utilizando código MOON– por ejemplo:

Código 5.3.12: Ejemplo con código MOON del análisis de **Generalizar acotación**

```
class Target[T -> B] inherit Ancestor1[T], Ancestor2[T]
signatures
    attributes
        p1 : Provider1[T];
        p2 : Provider2[T];
    -- etc.
body
end;

class A body end;

class B inherit A body end;

class C inherit B body end;

class D inherit C body end;

class Provider1[T -> A] body end;

class Provider2[T -> B] body end;

class Ancestor1[T -> A] body end;

class Ancestor2[T -> B] body end;

class Descendant1[T -> B] inherit Target[T] body end;

class Descendant2[T -> C] inherit Target[T] body end;

class Client1
signatures
    attributes
        a : Target[B]; -- client
body end;


class Client2
```

```
signatures
    attributes
        a : Target[C]; -- client
body end;

class Client3[S -> B]
signatures
    attributes
        a : Target[S]; -- client
body end;
```

El análisis se dirige en base a la caracterización propuesta previamente en el Capítulo 4, en la Sec. 4.2, en particular bajo la propiedad del ámbito de clase, cliente y herencia. Se supone que se generaliza la acotación por el tipo `A`.

Clase Al cambiar la acotación actual `B` de un parámetro formal por un nuevo tipo acotación `A` (en el ejemplo, en la clase `Target`, acotar `T ->A`), en primer lugar se debe cumplir una relación de subtipado entre el tipo de la acotación actual y el nuevo tipo acotación (`B <: A`). En otro caso no se garantiza que la semántica sea preservada.

Todo el conjunto de propiedades, utilizadas con entidades de la clase cuyo tipo es igual al parámetro genérico formal, deben estar contenidas en el nuevo tipo acotación. En caso contrario se impide la refactorización. La preservación del comportamiento se asegura en tanto en cuanto la redefinición de las propiedades en la jerarquía de herencia asegure la semántica dinámica. Esto puede ser comprobado en lenguajes que incorporan diseño por contrato como Eiffel, permitiendo el encolado de pre y postcondiciones en la redefinición de métodos. Sin embargo en lenguajes como Java, C# o C++ se delega básicamente en la comprobación de tipos, en argumentos y retorno básicamente, y suponiendo una implementación no maliciosa por parte del programador.

Cliente Las tipos paramétricos de clases **clientes**, que reemplazan al parámetro formal dependiente por posición (en el ejemplo en las clases `Client1, Client2` y `Client3`) deben tener inicialmente una acotación compatible con el cambio, es decir, subtipo de la nueva acotación (en el ejemplo si `B <: A` y `C <: B`, entonces `C <: A`). Siendo correcto el código previo a la refactorización, condición necesaria para iniciar la refactorización, generalizar la acotación *no tiene efectos sobre las clases clientes*, puesto que si inicialmente la sustitución en el cliente cumplía la acotación, seguirá cumpliendo al acotarse por un supertipo más general.

En todos los **proveedores** de la clase objetivo, el uso de entidades cuyo tipo está definido por un parámetro genérico formal dependiente debe ser analizado, comprobando que todas las propiedades utilizadas están definidas en el nuevo tipo acotación. Este punto se ve apoyado en la propia motivación de la refactorización (en el ejemplo en `Provider1` y `Provider2`).

Por otro lado, la acotación del parámetro formal dependiente en proveedores debe ser más general, nunca más estricta. Por lo tanto si el nuevo tipo acotación no es subtipo de la acotación del proveedor, el código generado sería incorrecto. Se debe repetir el proceso de análisis con el parámetro formal dependiente en el proveedor, con el fin de

generalizar su acotación. Pero si el nuevo tipo acotación es subtipo de la acotación actual del proveedor se finaliza el análisis, puesto que el código generado es válido sin modificación alguna.

Herencia La acotación en los parámetros formales dependientes en los **descendientes** seguirá siendo correcta. Dado que el parámetro formal en el descendiente cumplía la acotación del ancestro, al generalizarse dicha acotación en el ancestro, siempre se seguirá verificando que el descediente cumple la acotación. Así pues la refactorización *no tiene efectos sobre las clases descendientes.*

Por otro lado en relación a los **ancestros**, la acotación del parámetro formal dependiente debe ser más general, nunca más estricta. Por lo tanto:

- Si el nuevo tipo acotación no es subtipo de la acotación del ancestro, el código generado sería incorrecto. Se debe repetir el proceso de análisis con el parámetro formal dependiente en el ancestro, con el fin de generalizar su acotación.
- Si el nuevo tipo acotación es subtipo de la acotación del ancestro, se finaliza el análisis, puesto que el código generado es válido sin modificación alguna.

En nuestro ejemplo, en el Código 5.3.12, las clases clientes y descendientes (`Client1, Client2, Client3`, `Descendant1` y `Descendant2`) no son objeto del análisis a realizar. Sin embargo en el caso de los ancestros, en el ejemplo tanto `Ancestor1` como `Ancestor2`, y en el caso de los proveedores con `Provider1` y `Provider2`, sí son objeto de análisis.

Entradas

- Parámetro formal `formalPar : FormalPar`.
- Tipo acotación inicial `initialBoundType : ClassType`.
- Nuevo tipo acotación `newBoundType : ClassType`.

Precondiciones

Precondición 1: El tipo acotación inicial no puede ser un parámetro genérico formal. Esta restricción queda cubierta al establecer el tipo estático del argumento de entrada de la refactorización (`initialBoundType : ClassType`).

En caso de querer refactorizar una clase con parámetros formales acotados por otros parámetros formales de la propia clase, se debe aplicar primero la refactorización **Reemplazar parámetro formal por tipo completo** (ver Apartado 5.4.1).

Precondición 2: el nuevo tipo acotación es supertipo del tipo acotación actual. Se utiliza el predicado `IsSubtype`.

Código 5.3.13: Predicado `IsSubtype`

```
namespace Moon::Predicate;

activity IsSubtype(in subType : ClassType, in superType : ClassType) : Boolean
{
    return subType.isSubtype(superType);
}
```

Precondición 3: para todas las acotaciones en parámetros formales dependientes en ancestros y proveedores, las propiedades utilizadas están contenidas en el nuevo tipo acotación. Se utiliza el predicado `IsGeneralizable`.

Código 5.3.14: Predicado `IsGeneralizable`

```
namespace Moon::Predicate;

public import Moon::Function::DependantUpFormalPar;
public import Moon::Function::CollectEntitiesWithType;
public import Moon::Function::CollectPropertiesUsedByEntity;

activity IsGeneralizable(in formalPar : FormalPar, in newBoundType : ClassType) :
    Boolean
{
        // Aux lists, initially empty lists
        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        // Body

        for (FormalPar fp : DependantUpFormalPar(formalPar, visitedClasses,
            visitedFormalParameters)) {
                // for each entity with dependant type, for each property used by this
                    entity
                // the property is included in the new bound
                if (! (CollectEntitiesWithType(fp).toSequence()
                                -> forAll entity (CollectPropertiesUsedByEntity(entity)
                                    .toSequence()
                                        -> forAll prop (newBoundType.getClassDef().
                                            getBasicShortFlattened().toSequence()
                                                -> includes(prop))))) {
                        return false;
                } // if
        } // for

        // All properties check the condition
        return true;
}
```

Acciones

Acción: para todos los parámetros genéricos dependientes, directa o transitivamente, en proveedores y ancestros cuya acotación es subtipo de la nueva acotación dada, se modifica su acotación si fuera necesario. Se utiliza la acción **GeneralizeAction**.

Código 5.3.15: Acción `GeneralizeAction`

```
namespace Moon::Action;

public import Moon::Function::DependantUpFormalPar;
public import Moon::Function::ReplaceFormalPar;

activity GeneralizeAction(in initialFormalPar : FormalPar , in initialBoundType :
    ClassType, in newBoundType : ClassType) {

        // Aux lists, initially empty lists
        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        for (FormalPar fp : DependantUpFormalPar(initialFormalPar, visitedClasses,
            visitedFormalParameters)){
                if (((BoundS) fp).isBoundSubtype(newBoundType)){
                        ((BoundS) fp).getBounds().remove(initialBoundType);
                        ((BoundS) fp).getBounds().add(fp.replaceFormalPar(
                            initialFormalPar, newBoundType));
                }
        }
}
```

Por otro lado, también se utiliza el método `replaceFormalPar` de la clase `FormalPar` en el metamodelo MOON. Dicho método devuelve el tipo equivalente resultado de sustituir el nombre del parámetro formal objetivo de la refactorización, por el nuevo nombre del parámetro formal equivalente en la clase actual que se está refactorizando, especializando su acotación. Por ejemplo, si el parámetro formal inicial a especializar su acotación es `T` pero en la clase actual a refactorizar el parámetro formal equivalente es `S` acotado por `A<S>` –en aquellos casos donde existe acotación F– el nuevo tipo acotación no es `A<T>` sino `A<S>`. Este nuevo tipo acotación, que previamente pudiera no existir en el modelo, es calculado y obtenido a partir del método de utilidad `replaceFormalPar`.

Restricciones del lenguaje

Aplicable en lenguajes con acotación por subtipado o conformidad, incluyendo la acotación F. Aunque el análisis previo se ha realizado desde el punto de vista de subtipado, es igualmente aplicable utilizando la regla de conformidad en lugar de la regla de subtipado. Por ejemplo, utilizando el estudio previo presentado en la Sec. 2.2, la refactorización se puede aplicar en lenguajes como Java y C#, que utilizan acotación por subtipado, o bien en Eiffel que utiliza acotación por conformidad. En C++ esta refactorización no es aplicable al carecer de acotación en la definición de las plantillas (*templates*).

Refactorización

La refactorización **Generalizar acotación** queda finalmente construida de la siguiente forma expresada en Alf:

Código 5.3.16: Refactorización **Generalizar acotación**

```
namespace Moon::Refactoring;

public import Moon::Predicate::IsSubtype;
public import Moon::Predicate::IsGeneralizable;
public import Moon::Action::GeneralizeAction;

activity GeneralizeBoundRefactoring
        (in formalPar : FormalPar, in initialBoundType : ClassType, in newBoundType :
            ClassType) : Boolean
{
        // Pre:
        if (IsSubtype(initialBoundType, newBoundType) &&
                IsGeneralizable(formalPar, initialBoundType, newBoundType))
        {
                // Action:
                GeneralizeAction(formalPar, initialBoundType, newBoundType);
        }
        // Post:
        return true;
}
```

5.3.3. Refactorización Eliminar signaturas en cláusula *tal que (Remove signatures in where clause)*

Descripción

Eliminar las signaturas de aquellas propiedades que no son utilizadas, aún estando disponibles en todas las sustituciones actuales del parámetro formal (ver Fig. 5.4).

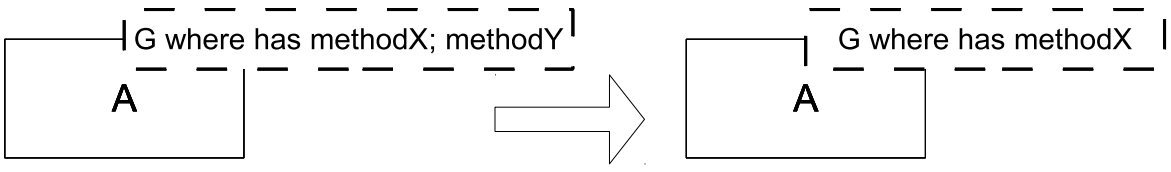

Figura 5.4: Eliminar signaturas en cláusula *tal que*

Motivación

En el contexto de una clase genérica, sólo se utiliza un subconjunto de las propiedades declaradas en la la acotación a partir de entidades cuyo tipo está definido por el parámetro formal. Eliminando estas propiedades en la acotación, que **restringen pero no son utilizadas**, se **generaliza** ampliando el conjunto posible de sustituciones al ser **menos estricta** la acotación.

Este problema es habitual cuando se ha intentado anticipar necesidades futuras que en la práctica nunca se han dado. El diseñador previó que se necesitarían ciertas propiedades como parte de la acotación, pero en la práctica no se han necesitado. Este problema es similar al producido en la acotación por subtipado con un tipo muy estricto, y sin embargo, las entidades cuyo tipo está definido por el parámetro formal sólo utilizan propiedades ya definidas en los ancestros de ese tipo acotación.

Análisis

Es fundamental que la propiedad eliminada no sea utilizada en la clase, a través de entidades cuyo tipo está definido por el parámetro genérico formal sobre el que se establece la acotación.

En los **proveedores** con parámetro dependiente, el método eliminado no debe pertenecer a la acotación del parámetro formal dependiente. En tal caso el parámetro formal en el proveedor no verificaría la acotación del parámetro en la clase que solicita servicios. Esta comprobación se debe repetir para los proveedores de los descendientes de la clase objetivo.

En los **ancestros** con parámetro dependiente, el conjunto de propiedades acotación se seguirá manteniendo, puesto que la acotación en el ancestro seguirá estando contenida en la nueva acotación. Suponiendo siempre que se produce herencia de las acotaciones entre ancestros y descendientes.

En los **descendientes** hay que comprobar que la propiedad eliminada no se utiliza por entidades cuyo tipo está definido por el parámetro formal dependiente.

Se podría plantear cierta interacción por parte del usuario a la hora de seleccionar las propiedades a eliminar. Pero se ha optado por, en una primera aproximación, definir una refactorización completamente automática en la que se eliminan de forma segura las propiedades de la cláusula que no son utilizadas.

Dada una clase con acotación por cláusula *tal que* y una serie de clases dependientes, por ejemplo:

Código 5.3.17: Ejemplo con código MOON del análisis de **Eliminar signaturas en cláusula tal que**

```
class A[T] where T has f, g
signatures
    attributes
        a1 : P[T]
        a2 : R[T]
        a3 : S[T]
body
end;

class B[T] where T has h inherit A[T]
-- signatures
body
end;

class P[T] where T has f
--signatures
body
```

```
end;

class R[T] where T has g
--signatures
body
end;

class S[T] where T has f, g
--signatures
body
end;
```

Si eliminamos la signatura `f` o `g` de la cláusula *tal que* en la clase `A`, se debe comprobar que ninguna entidad de tipo `T` en la clase `A` utiliza dichos métodos pudiéndose realizar la refactorización. En caso contrario, el código resultante sería incorrecto impidiendo la refactorización. Se debe repetir el análisis en el descendiente `B`, con el parámetro formal dependiente `T`, y recursivamente en los descendientes y los correspondientes proveedores de estas clases.

El análisis se dirige bajo el ámbito de clase, de cliente y de herencia.

Clase Se deben analizar las entidades cuyo tipo es el parámetro genérico formal de la clase, comprobando que ninguna entidad de tipo parámetro formal en la clase utiliza el método a eliminar.

Cliente En los clientes no hay efectos, pero en los proveedores se tiene que comprobar que la acotación es válida al eliminar el método, si existe en la acotación. Por lo tanto, para aquellas entidades del proveedor, cuyo tipo está definido por un parámetro formal dependiente, se debe comprobar que la propiedad nunca es invocada. Si no existe en la acotación no es necesario incluirlo en el análisis.

Herencia No tiene efectos sobre los ancestros. En los descendientes hay que tener en cuenta las entidades cuyo tipo es el parámetro genérico formal, comprobando que no se utiliza el método a eliminar. Recursivamente se extiende el análisis a proveedores y descendientes.

En este caso, como en las clases clientes y ancestros no hay efectos, no se realiza un análisis recursivo.

Entradas

Un parámetro formal con acotación con cláusula *tal que* `boundW : BoundW`.

Precondiciones

No se establecen precondiciones, salvo la implícita por la propia declaración de tipo del argumento de entrada. La restricción del tipado estático sobre la entrada de la refactorización impide que pueda ser iniciada sobre otro tipo de parámetro formal que no sea subtipo de `BoundW`.

Acciones

Acción: para el parámetro formal, se eliminan de la signatura de la cláusula *tal que* todos aquellos métodos que no son utilizados por entidades cuyo tipo es el parámetro formal.

Código 5.3.18: Acción `RemoveMethodsInWhereClause`

```
namespace Moon::Action;

public import Moon::Function::DependantDescendantsProvidersBoundW;
public import Moon::Function::CollectEntitiesWithType;
public import Moon::Function::CollectMethodsUsedByEntity;

activity RemoveMethodsInWhereClause(in boundW : BoundW) {
        List<Property> properties = boundW.getProperties(); // get bounds
        for (Property prop : properties) {
                List<Method> result = new List<Method>();
                List<ClassDef> visitedClasses = new List<ClassDef>();
                List<BoundW> visitedBoundWParameters = new List<BoundW>();
                DependantDescendantsProvidersBoundW(boundW, visitedClasses,
                        visitedBoundWParameters, (Method) prop);
                for (BoundW auxBoundW : visitedBoundWParameters) {
                        List<Entity> entities = CollectEntitiesWithType(auxBoundW);
                        for (Entity entity : entities) {
                                List<Method> methods = CollectMethodsUsedByEntity(
                                    entity);
                                result.addAll(methods);
                        }
                }
                if (!result.includes(prop)) { // not using the method
                        boundW.remove(prop); // remove the method
                }
        }
}
```

Postcondiciones

Postcondición: todos los métodos cuyas signaturas están incluidas en la cláusula *tal que*, son utilizados por alguna entidad cuyo tipo es uno de los parámetros genéricos dependientes.

Código 5.3.19: Predicado `ForAllEquivalentBoundWMethodsUsed`

```
namespace Moon::Predicate;

public import Moon::Function::DependantDescendantsProvidersBoundW;
public import Moon::Function::CollectMethodsUsedByEntity;
public import Moon::Function::CollectEntitiesWithType;

activity AllBoundWMethodsAreUsed(in boundW : BoundW) : Boolean
{
  return boundW.getProperties().toSequence()
        -> forAll prop (prop instanceof Method &&
          DependantDescendantsProvidersBoundW(boundW, visitedClasses,
                visitedBoundWParameters, (Method) prop).toSequence()
                  -> forAll bw (CollectEntitiesWithType(bw).toSequence()
```

```
                    -> exists entity (CollectMethodsUsedByEntity(entity)
                       -> includes(prop))));
}
```

Comentarios

En esta refactorización, así como en la refactorización en especialización **Añadir signaturas en cláusula tal que** (ver Apartado 5.4.3) se asume la herencia de las acotaciones, sin necesidad de volver a definir de nuevo la acotación en los descendientes. Por lo tanto, el efecto de eliminar o añadir un método a la cláusula *tal que* se aplica automáticamente en descendientes. Bajo el supuesto de que las acotaciones se heredan, se asume que no es correcto volver a declarar el método en la cláusula *tal que* de los descendientes.

Por otro lado, se considera que la refactorización no necesita interacción con el usuario, dado que la refactorización elimina los métodos no necesarios, se puede dar el caso de que ninguno de los métodos pueda ser eliminado, dejando el código en el mismo estado inicial.

Restricciones del lenguaje

Aplicable en lenguajes con acotación por cláusula *tal que* o cláusula *where*. En la actualidad, dentro de los lenguajes de la corriente principal, se restringe el ámbito de aplicación de esta refactorización a la acotación por el constructor sin argumentos presente en el lenguaje C#, o la acotación de constructores y tipos auto-inicializados en Eiffel [Ecma International, 2006].

Refactorización

La refactorización **Eliminar signaturas en cláusula tal que** queda finalmente construida de la siguiente forma expresada en Alf:

Código 5.3.20: Refactorización **Eliminar signaturas en cláusula tal que**

```
namespace Moon::Refactoring;

public import Moon::Predicate::AllBoundWMethodsAreUsed;
public import Moon::Action::RemoveMethodsInWhereClause;

activity RemoveSignaturesInWhereClausesRefactoring
        (in boundW : BoundW) : Boolean
{
        // Pre
        // Actions
        RemoveMethodsInWhereClause(boundW);
        // Post
        return AllBoundWMethodsAreUsed(boundW);
}
```

5.4. Catálogo en especialización

La tendencia por parte del desarrollador de software a intentar ser clarividente introduce problemas de un aumento innecesario de la complejidad. La anticipación de necesidades futuras, que nunca se llegan a dar en la práctica, es un problema difícil de evitar. Esto también es habitual ante nuevas utilidades en los lenguajes, que se introducen en el código por el mero hecho de ser novedosas.

En el caso concreto de la genericidad, la definición de parámetros formales sin restringir o con acotaciones débiles, abren la posibilidad de usar un amplio conjunto de tipos como sustitución. Pero, por otro lado, se impide el uso de propiedades que están disponibles en todas las sustituciones reales.

Las refactorizaciones cobran fuerza, en el contexto particular de la programación genérica, respecto a la especialización de la acotación de los parámetros genéricos. La especialización permite que el conjunto de propiedades a utilizar a través de entidades cuyo tipo está definido por el parámetro genérico sea más amplio, puesto que el número de ancestros en los caminos de herencia aumenta, y por lo tanto aumenta también el número de propiedades heredadas y accesibles.

A continuación se detalla el conjunto de refactorizaciones en genericidad respecto a la especialización de sus parámetros formales, de cara a sus posibles sustituciones.

5.4.1. Refactorización Reemplazar parámetro formal por tipo completo (*Replace formal parameter with complete type*)

El uso de la genericidad, con un cierto carácter de anticipación, puede llevar a la creación de clases genéricas en las que las sustituciones a un parámetro genérico formal son siempre por un mismo tipo. Esto no justifica el uso del polimorfismo paramétrico y del concepto de clase genérica. En otros casos, la necesidad de trabajar con versiones previas del lenguaje de programación, en las que no se incluía el concepto de clases genérica, puede motivar la eliminación de parámetros genéricos formales.

En tales casos, surge la necesidad de aplicar una operación en sentido contrario a la refactorización **Parametrización** (ver Apartado 5.3.1) que permita eliminar el uso del parámetro formal en favor de un tipo completo.

Descripción

Eliminar un parámetro formal en una clase por un tipo completo (ver Fig. 5.5).

Motivación

El parámetro formal es siempre reemplazado con el mismo tipo en todos los clientes y descendientes. El parámetro puede ser reemplazado con un tipo completo, y no es necesario el uso de mecanismos de conversión que incrementan la falta de fiabilidad en el código (como

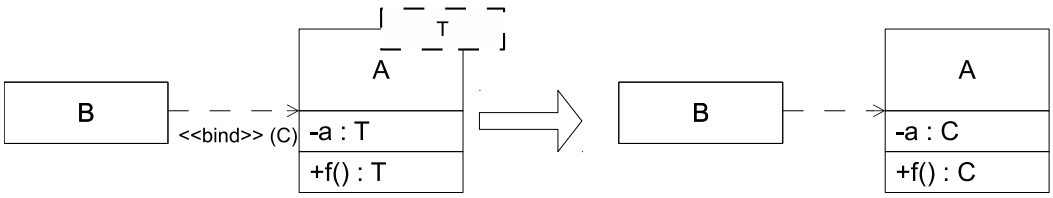

Figura 5.5: Reemplazar parámetro formal por tipo completo

por ejemplo el uso de *downcasts*). Este problema surge cuando la clase se ha creado con expectativas de ser usada de forma genérica, pero en la práctica sólo tiene un único uso. El diseñador intentó anticipar necesidades futuras que nunca han surgido posteriormente.

Análisis

Dada una clase con acotación (por subtipado o conformidad) y una serie de clases dependientes, por ejemplo:

Código 5.4.21: Ejemplo con código MOON del análisis de **Reemplazar parámetro formal por tipo completo**

```
class Target[T -> B] inherit Ancestor1[T], Ancestor2[T]
signatures
    attributes
        p1 : Provider1[T];
        p2 : Provider2[T];
    -- etc.
body
end;

class A body end;

class B inherit A body end;

class C inherit B body end;

class Provider1[T -> A] body end;

class Provider2[T -> B] body end;

class Ancestor1[T -> A] body end;

class Ancestor2[T -> B] body end;

class Descendant1[T -> B] inherit Target[T] body end;

class Descendant2[T -> C] inherit Target[T] body end;

class Descendant3[T -> C] inherit Target[B] body end;

class Client1
signatures
    attributes
```

```
        a : Target[B]; -- client
body end;

class Client2<S>
signatures
    attributes
        a : Target[S]; -- client
body end;
```

El análisis se dirige en base a la caracterización propuesta previamente en el Capítulo 4, en la Sec. 4.2, en particular bajo la propiedad del ámbito de clase, cliente y herencia.

Clase Si se quiere eliminar el parámetro formal `T` por un tipo completo (en el ejemplo `B`), este tipo completo debe ser una sustitución correcta a la acotación actual. Dado que se parte de un código correcto y que se ha verificado previamente que las sustituciones son por el mismo tipo completo, se verifica que es una sustitución correcta. En la clase `Target` se debe sustituir toda aparición del tipo `T` por el tipo completo.

Cliente En todos los **clientes** directos o indirectos, la sustitución dada como parámetro genérico actual al parámetro formal dependiente se corresponde con el tipo completo por el que se sustituye el parámetro genérico formal. Cualquier otra sustitución impide refactorizar, puesto que en tal caso la clase sí necesariamente se debe declarar como clase genérica. En caso de realizarse la refactorización, se debe eliminar la declaración paramétrica del tipo en relación al parámetro formal eliminado, como por ejemplo en `Client1` donde el atributo `a` pasa a ser de tipo `Target` en lugar de `Target[B]`.

Las tipos paramétricos de clases **proveedoras** en la clase `Target`, que utilizan en su declaración el parámetro formal `T` (ver clase `Provider1` y `Provider2`) deben sustituir el parámetro formal por el nuevo tipo completo en la instanciación genérica, en el ejemplo cambiando a `p1:Provider1[B]` y `p2:Provider[B]`. Dado que el parámetro `T` cumplía la acotación en estas sustituciones en las clases proveedoras y que el tipo completo utilizado como sustitución también verificaba dicha acotación, por lo tanto el tipo completo sigue siendo sustitución válida en los proveedores.

Herencia La acotación en los parámetros formales en los **ancestros** se mantiene sin cambios (ver `Ancestor1` y `Ancestor2`) y no introduce ninguna condición en su análisis que impida la refactorización. Dado que el tipo completo era sustitución válida del parámetro formal eliminado, es por lo tanto válido como sustitución del parámetro formal en los ancestros, puesto que seguirá verificando la acotación. Para llevar a cabo la refactorización se debe cambiar en la instanciación genérica el parámetro formal por el tipo completo, al igual que se realiza en los proveedores, teniendo finalmente `Ancestor1[B]` y `Ancestor2[B]`.

En los **descendientes** directos e indirectos con dependencia del parámetro formal de la clase (en el ejemplo descendientes de `Target`), se debe comprobar que no se dan como sustituciones tipos diferentes del tipo completo para poder realizar la refactorización. Si se cumple esta condición se elimina el parámetro formal sustituyendo dicho tipo en la clase por el tipo completo.

Entradas

- Parámetro formal `formalPar : FormalPar`.
- Tipo completo `ct : ClassType`.

Precondiciones

Precondición 1: El tipo dado como sustitución es un tipo completo (`ct : ClassType`) no dependiendo en su definición de ningún parámetro formal de la clase donde se utiliza. Se utiliza el predicado `IsComplete`.

Código 5.4.22: Predicado `IsComplete`

```
namespace Moon::Predicate;

activity IsComplete(in type : ClassType) : Boolean
{
        return type.isComplete(); // in MOON metamodel
}
```

Precondición 2: Todas las sustituciones del parámetro formal en clases clientes y descendientes, directa o transitivamente, dan como parámetro actual el mismo tipo completo (`ct : ClassType`). Se utiliza el predicado `ForAllSubstitutionsSameType`.

Código 5.4.23: Predicado `ForAllSubstitutionsSameType`

```
namespace Moon::Predicate;

public import Moon::Function::DependantDownFormalPar;
public import Moon::Function::RealSubstitutionsOfFormalPar;

activity ForAllSubstitutionsSameType(in formalPar : FormalPar, in newBoundType :
    ClassType) : Boolean
{
        // Aux lists, initially empty lists
        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        return (DependantDownFormalPar(formalPar, visitedClasses,
            visitedFormalParameters).toSequence() -> forAll fp
                        (RealSubstitutionsOfFormalPar(fp) -> forAll t (t ==
                            newBoundType) ));
}
```

Precondición 3: Todas las acotaciones del parámetro formal en clases clientes y descendientes, directa o transitivamente, tiene como acotación el mismo tipo completo (`ct : ClassType`). Se utiliza el predicado `ForAllBoundsSameType`.

Código 5.4.24: Predicado `ForAllBoundsSameType`

```
namespace Moon::Predicate;

public import Moon::Function::DependantDownFormalPar;

activity ForAllBoundsSameType(in formalPar : FormalPar, in classType : ClassType) :
    Boolean
{
        // Aux lists, initially empty lists
        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        return (DependantDownFormalPar(formalPar, visitedClasses,
            visitedFormalParameters).toSequence() -> forAll fp
                        (((BoundS) fp).getBounds() -> forAll t (t == classType) ));
}
```

La restricción añadida con esta precondición, aún siendo muy restrictiva, se justifica con el fin de asegurar la preservación del comportamiento. Aunque pudiera darse la situación de que no existen sustituciones actuales del parámetro acotado por el tipo completo que impidan la refactorización, si se permitiese realizar la refactorización se permitiría en un futuro sustituir por tipos completos que no son el mismo subtipo por el que se ha sustituido en la clase objetivo de la refactorización, y por lo tanto con una semántica diferente.

En el ejemplo, si no hay instanciaciones genéricas de `Descendant2` o de `Client2`, donde la sustitución del parámetro formal dependiente sea un tipo completo (por ejemplo) `C`, la **Precondición 2** se verificaría, pero en un futuro uso de la clase se estaría utilizando un tipo `B` (ahora directamente codificado en `Target`) cuando por el uso de la genericidad se estaría utilizando en el código previo a la refactorización un tipo `C` en la clase `Target`, no garantizando el mismo comportamiento.

Acciones

Acción 1: En el uso de los tipos definidos por los ancestros y proveedores de aquellas clases en las que se debe eliminar el parámetro formal (parámetros formales dependientes), se reemplaza el parámetro formal dependiente en las declaraciones de tipos por el tipo completo (`ct:ClassType`). Se define la acción `ReplaceFormalParInAncestorsAndProvidersAction`.

Esta acción se apoya en el método `calculateType` de la clase `Type` implementado en el metamodelo MOON. Dicho método genera un nuevo tipo resultado de sustituir en el tipo actual el parámetro formal, por el tipo actual pasado como argumento.

Por ejemplo, suponiendo que el tipo pasado como argumento es `A` y el parámetro formal es `T` se tendrían los siguientes resultados, suponiendo `C` como tipo paramétrico:

- aplicar el método sobre el tipo `T` genera el tipo `A`.

Código 5.4.25: Acción `ReplaceFormalParInAncestorsAndProvidersAction`

```
namespace Moon::Action;

public import Moon::Function::DependantDownFormalPar;

activity ReplaceFormalParInAncestorsAndProvidersAction(in initialFormalPar : FormalPar,
     in classType : ClassType) {

        // Aux lists, initially empty lists
        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        // Aux variables
        List<InheritanceClause> inheritanceClauseList = new List<InheritanceClause>();
        List<Entity> entitiesInClass = new List<Entity>();

        // Process all formal dependant parameters...
        for (FormalPar fp : DependantDownFormalPar(initialFormalPar, visitedClasses,
            visitedFormalParameters)) {
                // Replace in type of direct ancestors in inheritance clauses
                inheritanceClauseList = fp.getClassDef().getInheritanceClause();
                for(InheritanceClause ic : inheritanceClauseList) {
                        if (ic.getType().getRelatedFormalParameters() -> includes(fp))
                            {
                                Type type = ic.getType();
                                ic.setType(type.calculateType(classType, fp));
                        }
                }
                // Replace in type of entities in direct clients
                entitiesInClass = fp.getClassDef().getEntities();
                for(Entity entity : entitiesInClass) {
                        if (entity.getType().getRelatedFormalParameters() -> includes(
                            fp)) {
                                entity.setType(entity.getType().calculateType(classType
                                    , fp));
                        }
                }
        }
}
```

- aplicar el método sobre el tipo `C[T]` genera el tipo `C[A]`.
- aplicar el método sobre el tipo `C[C[T]]` genera el tipo `C[C[A]]`.

Acción 2: En las clases con parámetros formales dependientes se elimina el parámetro formal.

Código 5.4.26: Acción `RemoveFormalParInDependantDownAction`

```
namespace Moon::Action;

public import Moon::Function::DependantDownFormalPar;
```

```
activity RemoveFormalParInDependantDownAction(in initialFormalPar : FormalPar, in
    classType : ClassType) {

        // Aux lists, initially empty lists
        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        // Process all formal dependant parameters...
        for (FormalPar fp : DependantDownFormalPar(initialFormalPar, visitedClasses,
            visitedFormalParameters)) {
                fp.getClassDef().remove(fp); // remove formal parameter in class
        }
}
```

Restricciones del lenguaje

Aplicable con lenguajes de programación que incluyan programación genérica con acotación por subtipado/conformidad como Java, C# o Eiffel.

Refactorización

La refactorización **Reemplazar parámetro formal por tipo completo** queda finalmente construida de la siguiente forma expresada en Alf:

Código 5.4.27: Refactorización **Reemplazar parámetro formal por tipo completo**

```
namespace Moon::Refactoring;

public import Moon::Predicate::IsComplete;
public import Moon::Predicate::ForAllSubstitutionsSameType;
public import Moon::Predicate::ForAllBoundsSameType;
public import Moon::Action::ReplaceFormalParInAncestorsAndProvidersAction;
public import Moon::Action::RemoveFormalParInDependantDownAction;

activity ReplaceFormaParWithCompleteTypeRefactoring
        (in formalPar : FormalPar, in classType : ClassType) : Boolean
{
        // Pre:
        if (IsComplete(classType) &&
                ForAllSubstitutionsSameType(formalPar, classType) &&
                ForAllBoundsSameType(formalPar, classType))
        {
                // Action:
                ReplaceFormalParInAncestorsAndProvidersAction(formalPar,classType);
                RemoveFormalParInDependantDownAction(formalPar, classType);
        }
        // Post:
        return true;
}
```

5.4.2. Refactorización Especializar acotación (*Specialize bound*)

Descripción

Reemplazar el tipo acotación con uno de sus subtipos (ver Fig. 5.6).

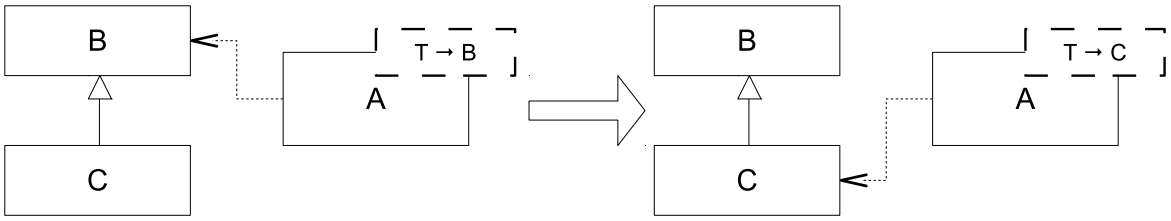

Figura 5.6: Especializar acotación (*Specialize bound*)

Motivación

Todas las sustituciones actuales del parámetro formal son subtipos de la nueva acotación que se propone, permitiendo, al cambiar la acotación, utilizar un conjunto más amplio de propiedades en un futuro.

Análisis

Dada una clase con acotación por subtipado y una serie de clases dependientes –utilizando código MOON– por ejemplo:

Código 5.4.28: Ejemplo con código MOON del análisis de **Especializar acotación**

```
class Target[T -> B] inherit Ancestor1[T], Ancestor2[T]
signatures
    attributes
        p1 : Provider1[T];
        p2 : Provider2[T];
    -- etc.
body
end;

class B inherit A body end;

class C inherit B body end;

class D inherit C body end;

class Provider1[T -> A] body end;

class Provider2[T -> B] body end;
```

```
class Ancestor1[T -> A] body end;

class Ancestor2[T -> B] body end;

class Descendant1[T -> C] inherit Target[T] body end;

class Descendant2[T -> D] inherit Target[T] body end;
```

El análisis se dirige en base a la caracterización propuesta previamente en el Capítulo 4, en la Sec. 4.2, en particular bajo el ámbito de clase, cliente y herencia. Se supone que se especializa la acotación por el tipo `C`.

Clase Si se quiere cambiar la acotación `B` de un parámetro formal por un tipo más estricto, en primer lugar se debe cumplir una relación de subtipado entre el nuevo tipo acotación y el tipo de la acotación actual. En otro caso no se garantiza que la semántica sea preservada.

Todas las propiedades de entidades dependientes en tipo del parámetro formal en la clase, seguirán estando contenidas en la nueva acotación, garantizando la corrección del código en la clase. Si la redefinición de propiedades cumple la semántica de la propiedad inicial se garantiza la preservación del comportamiento.

Cliente Las tipos paramétricos de clases **proveedoras**, que utilizan en su declaración el parámetro formal o parámetro dependiente por posición (en el ejemplo, las clases `Provider1` y `Provider2`) deben tener inicialmente una acotación compatible con el cambio (supertipo de la nueva acotación, en el ejemplo si `B <: A` y `C <: B`, entonces `C <: A`). Siendo correcto el código previo a la refactorización (condición necesaria para iniciar la refactorización) *no tiene efectos sobre las clases proveedoras.*

En todos los **clientes** de la clase objetivo, las sustituciones del parámetro formal deben ser subtipos de la nueva acotación, en caso contrario la refactorización no puede llevarse a cabo. Este punto se ve apoyado en la propia motivación de la refactorización.

Herencia La acotación en los parámetros formales en los **ancestros**, es menos estricta, por lo tanto se garantiza que el nuevo tipo acotación es subtipo de la acotación del ancestro. De forma similar a como ocurre con los proveedores, la refactorización *no tiene efectos sobre las clases ancestro.*

La acotación del parámetro formal dependiente los **descendientes**, puede ser más estricta, nunca menos. Por lo tanto:

- Si el nuevo tipo acotación NO es supertipo de la acotación del descendiente, se debe repetir el proceso de análisis para intentar especializarlo también, con el parámetro formal dependiente y su acotación.
- Si el nuevo tipo acotación es supertipo de la acotación del descendiente, se finaliza el análisis.

En nuestro ejemplo, en el Código 5.4.28, las clases proveedoras y ancestros (`Provider1`, `Provider2`, `Ancestor1` y `Ancestor2`) no son objeto del análisis a realizar. Sin embargo, en el caso de los descendientes, en el ejemplo tanto `Descendant1` como `Descendant2`, las acotaciones de los parámetros formales dependientes por posición (`C` y `D`) son subtipo del nuevo tipo acotación, `C` en este caso, finalizando el análisis de ambas clases. Sin embargo, si al especializar la acotación de la clase objetivo el nuevo tipo acotación fuese `D` en lugar de `C`, entonces sobre la clase cliente `Descendant1` se debería repetir el análisis con el objetivo de especializar su acotación.

Entradas

- Parámetro formal `formalPar : FormalPar`.
- Tipo acotación inicial `initialBoundType : ClassType`.
- Nuevo tipo acotación `newBoundType : ClassType`.

Precondiciones

Precondición 1: El tipo acotación inicial no puede ser un parámetro genérico formal. Esta restricción queda cubierta al establecer el tipo estático del argumento de entrada de la refactorización (`initialBoundType : ClassType`).

En caso de querer refactorizar una clase con parámetros formales acotados por otros parámetros formales de la propia clase, se debe aplicar primero la refactorización **Reemplazar parámetro formal por tipo completo** (ver el Apartado 5.4.1).

Precondición 2: el nuevo tipo acotación es subtipo del tipo acotación actual. Se utiliza el predicado `IsSubtype` (ver Código 5.3.13).

Precondición 3: para todas las acotaciones en parámetros formales dependientes en descendientes y clientes, sus sustituciones son compatibles (subtipos) con el nuevo tipo acotación. Se utiliza el predicado `IsSpecializable`.

Código 5.4.29: Predicado `IsSpecializable`

```
namespace Moon::Predicate;

activity IsSpecializable(in formalPar : FormalPar, in newBoundType : ClassType) :
    Boolean
{
    // Aux lists, initially empty lists
    List<ClassDef> visitedClasses = new List<ClassDef>();
    List<FormalPar> visitedFormalParameters = new List<FormalPar>();

    return (DependantDownFormalPar(formalPar, visitedClasses, visitedFormalParameters).
        toSequence() -> forAll fp
                        (RealSubstitutionsOfFormalPar(fp) -> forAll ct (ct.isSubtype(
                            newBoundType))));
}
```

Acciones

Acción: para todos los parámetros genéricos dependientes, directa o transitivamente, en clientes y descendientes cuya acotación es supertipo de la nueva acotación dada, se modifica su acotación. Se utiliza la acción **SpecializeAction**.

Código 5.4.30: Acción `SpecializeAction`

```
namespace Moon::Action;

activity SpecializeAction(in initialFormalPar : FormalPar , in initialBoundType :
    ClassType, in newBoundType : ClassType) {

        // Aux lists, initially empty lists
        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        List<FormalPar> filteredFormalParameters =
                DependantDownFormalPar(initialFormalPar, visitedClasses,
                    visitedFormalParameters)
                        -> select fp (
                        ((BoundS) fp).isBoundSupertype(newBoundType)
                        && !((BoundS) fp).getBounds() -> contains(newBoundType));

        for (FormalPar fp : filteredFormalParameters) {
                        // the bound must be specialized
                        List<Type> boundsOfFormalPar = ((BoundS) fp).getBounds()->
                                excluding(initialBoundType);

                        boundsOfFormalPar -> removeAll (boundsOfFormalPar ->
                                select bound (newBoundType.isSubtype(bound)));

                        boundsOfFormalPar -> including(initialFormalPar.
                            replaceFormalPar(fp, newBoundType));
        } // for
}
```

En la especificación se utiliza el método `isBoundSupertype` de la clase `BoundS` en el metamodelo MOON. El método comprueba si alguno de los tipos acotación del parámetro formal es supertipo del tipo pasado como argumento de entrada. Por otro lado, se vuelve a utilizar el método `replaceFormalPar` de la clase `FormalPar` en el metamodelo MOON.

Restricciones del lenguaje

Aplicable en lenguajes con acotación por subtipado o conformidad, incluyendo la acotación F, como Java y C#. Aunque el análisis previo se ha realizado desde el punto de vista de subtipado, es igualmente aplicable utilizando la regla de conformidad en lugar de la regla de subtipado, como Eiffel. En C++ esta refactorización no es aplicable al carecer de acotación en la definición de las plantillas (*templates*).

Refactorización

La refactorización **Especializar acotación** queda finalmente construida de la siguiente forma expresada en Alf:

Código 5.4.31: Refactorización **SpecializeBoundRefactoring**

```
namespace Moon::Refactoring;

activity SpecializeBoundRefactoring
        (in formalPar : FormalPar, in initialBoundType : ClassType, in newBoundType :
            ClassType) : Boolean
{
        if (
        // Pre:
        IsSubtype(newBoundType, initialBoundType) &&
        // Pre:
        IsSpecializable(formalPar, initialBoundType, newBoundType))
        ){
                // Action:
                SpecializeAction(formalPar, initialBoundType, newBoundType);
        }
        // Post:
        return true;
}
```

5.4.3. Refactorización Añadir signaturas en cláusulas *tal que (Add signatures in where clause)*

Descripción

Añadir las signaturas de aquellos métodos que están disponibles en todas las ***sustituciones actuales*** *de un parámetro formal, a su acotación por cláusulas* ***tal que*** *o* **where clauses** *(ver Fig. 5.7).*

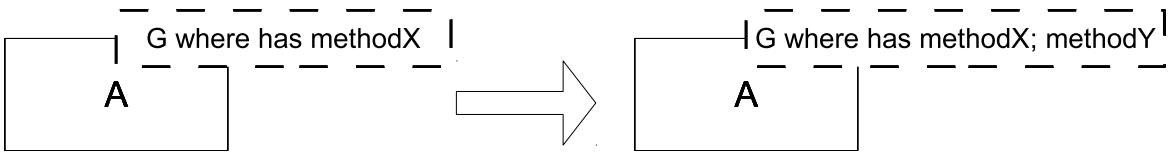

Figura 5.7: Añadir signaturas en cláusula *tal que*

Motivación

Cuando todos los tipos definidos por clases dados como sustitución a un parámetro formal de una clase genérica, tienen en común uno o más métodos y en las modificaciones previstas

para dicha clase genérica se hace necesario utilizar dicho(s) método(s), se puede añadir a la acotación las signaturas de dichos métodos en la cláusula *tal que*.

Esto especializa el conjunto de sustituciones reales a dicho parámetro, permitiendo el uso de dichas propiedades a través de las entidades, cuya declaración de tipo es el parámetro formal. En contraposición a la refactorización **Eliminar signaturas en cláusula tal que** (ver Apartado 5.3.3), esta refactorización cubre necesidades reales no previstas inicialmente. Estas refactorizaciones están disponibles y pueden ejecutarse con una mínima interacción del usuario, simplemente indicando como entrada el parámetro formal. El usuario podría requerir aplicar una u otra en función de la evolución necesaria en sus tareas de mantenimiento del código fuente. En ambos casos se evitan los errores fruto de la introducción o borrado de signaturas de manera manual.

En algunos lenguajes será necesario incluir en el análisis la accesibilidad de las propiedades (según los modificadores de acceso, listas de exportación, etc.), pero esta característica es particular de cada lenguaje de POO, y por lo tanto será resuelto en la implementación concreta de la extensión del metamodelo para el mismo.

Análisis

Dada una clase con acotación por cláusulas *tal que* y una serie de clases dependientes, obtenidas todas las sustituciones actuales del parámetro T, así como de los parámetros formales dependientes en otras clases (**clientes** y **descendientes** recursivamente), el conjunto intersección de todas las propiedades que están definidas en dichas clases, es igual a la acotación más estricta que se puede realizar sobre el parámetro formal y parámetros dependientes en clientes y descendientes.

Los **proveedores** no deben sufrir cambios puesto que el parámetro formal seguirá conteniendo un conjunto de propiedades dentro de la acotación establecida en el proveedor que no sufre cambios (el conjunto de propiedades se amplía, no se reduce preservando el cumplimiento de la acotación en el proveedor).

Igualmente en los **ancestros**, el conjunto de propiedades se seguirá manteniendo, puesto que la acotación tampoco sufre cambios, se conserva la acotación inicial añadiendo en todo casos nuevos métodos.

Se podría plantear cierta interacción por parte del usuario a la hora de añadir alguna o todas las nuevas propiedades, pero en una primera aproximación automática se añadirían todas.

En el ejemplo mostrado en el Código 5.4.32 y dados los siguientes casos (Código 5.4.33 y Código 5.4.34), en los que las clases `A` y `B` tienen diferentes conjuntos de métodos declarados, se plantean dos situaciones.

Código 5.4.32: Ejemplo con código MOON del análisis de **Añadir signaturas en cláusula tal que**

```
class Target1[T] where T has m1()
-- signatures
body end;

class Target2[T] where T has m2() inherit Target1[T]
-- signatures
body end;

class Client
signatures
    attributes
        client1 : Target1[A]; -- A provides m1
        client2 : Target2[B]; -- B provides m1 and m2
body
end;
```

Código 5.4.33: Añadir signaturas en cláusula *tal que*: Situación a)

```
class A
    signatures
        methods
            m1;
            m3;
body end;

class B
    signatures
        methods
            m1;
            m2;
            m3;
body end;
```

Código 5.4.34: Añadir signaturas en cláusula *tal que*: Situación b)

```
class A
    signatures
        methods
            m1;
            m3;
body end;

class B
    signatures
        methods
            m1;
            m2;
body end;
```

En la situación a) Se podría añadir la acotación `m3` porque todas las sustituciones en clases clientes tienen dicha propiedad. Por el contrario, en la situación b) no sería correcto porque la posterior sustitución en clientes de `Target2[B]` incumpliría la acotación al no tener declarado el método `m3`.

El análisis se dirige bajo el ámbito de clase, de cliente y de herencia.

Clase No es necesario realizar ningún análisis de la clase puesto que la adición de nuevas signaturas no afecta a la corrección de la misma, salvo por las relaciones de cliente y herencia.

Cliente En todos los clientes, las sustituciones reales del parámetro formal deben tener como método disponible aquel que se añada en la cláusula *tal que*, en caso contrario la sustitución pasaría a ser inválida. Los proveedores no son objeto de análisis.

Herencia En los descendientes hay que tener en cuenta los parámetros formales dependientes para repetir de nuevo el análisis recursivamente respecto a los clientes y descendientes. Los ancestros no son objeto de análisis.

Entradas

Un parámetro formal con acotación con cláusula *tal que* `boundW : BoundW`.

Precondiciones

No se establecen precondiciones, salvo la implícita por la propia declaración de tipo del argumento de entrada. La restricción del tipado estático sobre la entrada de la refactorización impide que pueda ser iniciada sobre otro tipo de parámetro formal que no sea subtipo de `BoundW`.

Acciones

Acción: para el parámetro formal de entrada, se añaden a la cláusula *tal que* todos los métodos comunes a las sustituciones. La recolección de los métodos a añadir se hace a través de la función `CollectSignaturesToAdd`.

Código 5.4.35: Acción `AddMethodsInWhereClause`

```
namespace Moon::Action;

/*
 * Add the set of methods in where clause.
 */
activity AddMethodsInWhereClause(in boundW : BoundW, in methods:List<Method>) {
        boundW.getProperties().addAll(methods);
}
```

Postcondiciones

Postcondición: todos los métodos en la cláusula tal que están incluidos en sustituciones reales del parámetro formal y parámetros dependientes. Se utiliza el predicado `ForAll-SignatureIsIncludedInSubstitutions`.

Código 5.4.36: Predicado `ForAllSignatureIsIncludedInSubstitutions`

```
namespace Moon::Predicate;

public import Moon::Function::CollectDownRealSubstitutions;

/*
 * All methods in where clause are included in real substitutions.
 */
activity ForAllSignatureIsIncludedInSubstitutions(in boundW : BoundW) : Boolean
{
        return boundW.getProperties().toSequence()
                -> forAll prop (prop instanceof Method &&
                                        CollectDownRealSubstitutions(boundW).
                    toSequence() ->
                            forAll ct (ct.getClassDef().getBasicShortFlattened().
                                toSequence() ->
                                    includes(prop)));
}
```

Se utiliza el operador `includes` de ALF dado que no existe en la especificación del lenguaje una operación de tipo *match* o de correspondencia por tipos, aunque en una implementación posterior dependiendo del lenguaje de implementación se deba implementar como una operación *match*.

Por otro lado, se utiliza el método `getBasicShortFlattened()` de la clase `ClassDef` en el metamodelo MOON que devuelve el conjunto de propiedades accesibles a través de herencia disponibles en la clase.

Comentarios

La refactorización presentada es una variación de la refactorización **Add Where Clause** y de la refactorización **Add Signature in Where Clause** presentada en [Marticorena et al., 2003]. En el primer caso se suponía que cuando el parámetro formal no tiene definida ninguna cláusula *tal que*, la cláusula es añadida automáticamente al añadir las correspondientes signaturas inferidas en la acotación. En el segundo caso, se supone que la nueva signatura a añadir a la acotación actual no es una entrada de la refactorización, sino inferida de las sustituciones actuales del parámetro formal. Si la signatura ya estaba incluida en la acotación, simplemente no se añadiría.

Por motivos de simplicidad y reutilización se han fusionado ambas refactorizaciones sin hacer distinciones si el parámetro formal ya establecía o no acotación previa con signaturas de métodos.

Restricciones del lenguaje

Aplicable en lenguajes con acotación por cláusulas *tal que* o cláusula *where*. En la actualidad, dentro de los lenguajes de la corriente principal, se restringe a la acotación en los constructores

por el constructor sin argumentos en el lenguaje C# o la acotación de constructores y tipos auto-inicializados en EIFFEL [Ecma International, 2006].

Refactorización

La refactorización **AÑADIR SIGNATURAS EN CLÁUSULA TAL QUE** queda finalmente construida de la siguiente forma expresada en el Código 5.4.37 en ALF:

Código 5.4.37: Refactorización **AÑADIR SIGNATURAS EN CLÁUSULA TAL QUE**

```
namespace Moon::Refactoring;

public import Moon::Function::CollectSignaturesToAdd;
public import Moon::Action::AddMethodsInWhereClause;

activity AddSignaturesInWhereClausesRefactoring
        (in boundW : BoundW) : Boolean
{
        // Pre
        // Actions
        AddMethodsInWhereClause(boundW, CollectSignaturesToAdd(boundW));
        // Post
        return ForAllSignatureIsIncludedInSubstitutions(boundW);
}
```

5.5. Caracterización del catálogo de refactorizaciones en genericidad

Una vez presentado el catálogo de refactorizaciones en genericidad, a lo largo de las secciones previas, se vuelven a plantear las cuestiones habituales a la hora de abordar su implementación. Cuestiones como las descritas en la Sec. 4.2, pueden y deben ser respondidas a través de caracterizaciones.

Partiendo de la caracterización propuesta en la Apartado 4.2.1, se obtiene la siguiente ordenación (ver Tabla 5.1) en relación al catálogo de refactorizaciones descrito. Tal y como se ha detallado, la caracterización permite por un lado establecer el orden de implementación, basándonos en la complejidad de la refactorización. Por otro lado permite abordar cuestiones relativas a la reutilización de interfaces gráficas según el conjunto de entradas.

Como se puede observar en la Tabla 5.2, las refactorizaciones **GENERALIZAR ACOTACIÓN** (ver Apartado 5.3.2), **ESPECIALIZAR ACOTACIÓN** (ver Apartado 5.4.2) y **REEMPLAZAR PARÁMETRO FORMAL POR TIPO** (ver Apartado 5.4.1), usarían la misma interfaz de usuario al tener el mismo conjunto de entradas.

Por otro lado, las refactorizaciones **AÑADIR SIGNATURA EN CLÁUSULAS TAL QUE** (ver Apartado 5.4.3) y **ELIMINAR SIGNATURA EN CLÁUSULA TAL QUE** (Apartado 5.3.3) tienen igual interfaz, siendo el único caso particular **PARAMETERIZE** (ver Apartado 5.3.1), puesto que tiene un conjunto de entradas diferente al resto de refactorizaciones del catálogo.

Sin embargo, se debe señalar que el tamaño reducido del catálogo presentado limita la utilidad de aplicar la caracterización propuesta, frente a su aplicación sobre un catálogo mayor. En el Apéndice B, se encuentra la aplicación al catálogo de refactorizaciones de Fowler compuesto por 68 refactorizaciones.

Tabla 5.2: Orden parcial de mayor a menor complejidad del grupo de refactorizaciones en genericidad

	Refactoring	Scope	DLI	Root input	Additional Inputs	Action
1	Generalize Bound	ICH	Advanced	Formal Parameter	* Classes	Replace
2	Specialize Bound	ICH	Advanced	Formal Parameter	* Classes	Replace
3	Replace Formal Parameter with Type	ICH	Advanced	Formal Parameter	* Class	Replace
4	Remove Signature in Where Clause	ICH	Advanced	Formal Parameter	No Additional Inputs	Remove
5	Add Signature in Where Clause	ICH	Advanced	Formal Parameter	No Additional Inputs	Add
6	Parameterize	ICH	Advanced	Entity	1 Name	Replace

CAPÍTULO 6

SOLUCIÓN BASADA EN FRAMEWORKS

En este capítulo se muestra nuestra solución de diseño para soportar un enfoque de refactorización con cierta independencia del lenguaje. El objetivo es permitir trabajar desde un doble punto de vista: con elementos independientes del lenguaje, definidos sobre el metamodelo MOON, y con elementos particulares del lenguaje objetivo. Para ello, una vez definidas y validadas las refactorizaciones, se da una propuesta de solución basada en *frameworks* para su soporte. La solución presentada debe permitir generalizar a partir de soluciones abstractas, dado el objetivo fundamental para el que se diseñó el metamodelo MOON. Por otro lado, también debe poderse especializar para un lenguaje objetivo concreto para el que se refactoriza, permitiendo construir y ejecutar refactorizaciones reales a través de herramientas o entornos de desarrollo integrados.

Este problema puede resolverse de manera natural con el concepto de *framework* y las instanciaciones del mismo. Si en la solución final propuesta se puede reutilizar el control de flujo del *framework* para cada una de sus diferentes instanciaciones, se acabarán obteniendo grandes beneficios desde el punto de vista de la reutilización.

En lo que sigue, la Sección. 6.1 introduce el concepto de *framework*, tal y como se entiende en el resto del trabajo, con sus beneficios, problemas, clasificación y características particulares respecto a su documentación. La Sec. 6.2 describe las distintas soluciones para representar el código para su consulta y manipulación. La Sec. 6.3 prosigue exponiendo las diferentes formas que se establecen para refactorizar el código. Finalmente, en la Sec. 6.4 se muestra cómo se recupera el código refactorizado, y se finaliza este capítulo en la Sección. 6.5 con las conclusiones obtenidas.

6.1. *Frameworks* como solución

Dada la variedad de acepciones que tiene en el mundo de la informática el término *framework*, en esta sección se establece claramente el término, tal y como será entendido en el resto del trabajo. También se enuncian los beneficios y problemas que origina su uso, así como la clasificación y guías de documentación a utilizar.

6.1.1. Concepto de *framework*

Incluso fuera del mundo del software y de la ingeniería del software, el término *framework* denota diferentes conceptos. Estos van desde una estructura que soporta algo o el marco o estructura compuesta de partes ajustadas e integradas. En todos los casos, el concepto de estructura para la resolución de un problema es válido para nuestro contexto de refactorización del software. Así pues, como primera definición fundamental, un *framework* se considera como un conjunto de elementos, que juntos forman una estructura que apoya la resolución de problemas.

Sin embargo siguiendo la definición previa, dentro del mundo de la informática, el término se utiliza de distinta manera dependiendo del ámbito de aplicación: como sinónimo de la arquitectura software, como metodología o proceso seguido, como herramienta utilizada, como conjunto de lenguajes de modelado y programación, etc. Un ejemplo claro se tiene la en la adopción del término por parte del *framework* .NET [Troelsen, 2001, Duffy, 2006], que incluye a su vez muchos otros conceptos.

En contraposición, en el presente trabajo se tomará como base el concepto de *framework orientado a objetos*[1]. Tomando la definición de dicho concepto de [Fayad et al., 1999], tal y como será utilizado a lo largo de este trabajo. Se define *framework* como:

> *"Diseño reutilizable de un sistema que describe* ***cómo*** *el sistema es descompuesto en un conjunto de objetos que* ***interactúan****."*

El *framework* describe la interfaz de cada objeto y el flujo de control entre los objetos marcando las responsabilidades asignadas a cada uno. Estos permiten generar aplicaciones en el ámbito de un dominio software para una familia de aplicaciones, en nuestro caso particular para la refactorización del software.

Por un lado, se tienen unas partes fijas, denominadas *frozen-spots*, donde realmente se produce la inversión de control, y desde donde se utilizan las extensiones de la instanciación concreta. Esta parte debe ser plenamente reutilizable para diferentes aplicaciones en el mismo dominio, a través de los métodos plantilla (**Template Method**) [Gamma et al., 1995]. Estos definen de forma concreta la lógica de interacción y el control de flujo en el *framework*.

Por otro lado, es vital introducir puntos de flexibilidad y extensión en el *framework*, denominados *hot-spots*, de tal forma que se puedan modificar los requisitos particulares, ajustando el uso del *framework*.

[1] En lo que sigue, se utilizará el término abreviado de *framework*.

Los *hot-spots* se construyen a partir de clases y métodos abstractos. Se definen los denominados métodos de enganche (*hook methods*), que cumplen el papel de puntos calientes flexibles, invocados en los métodos plantilla. Los métodos de enganche deben ser implementados de manera concreta para introducir la nueva funcionalidad. A partir de su implementación, se produce posteriormente una instanciación y puesta en ejecución del *framework*.

6.1.2. Ventajas y desventajas del uso de *frameworks*

Ventajas

Modularidad se promueve la modularidad, encapsulando los detalles de implementación más volátiles tras interfaces estables. Localiza el impacto de los cambios de diseño e implementación, reduciendo el esfuerzo requerido para comprender y mantener el código existente.

Reutilización se definen componentes genéricos que pueden ser aplicados en la creación de nuevas aplicaciones.

Extensibilidad a través de los métodos de enganche (*hook methods*) se permite a las aplicaciones extender sus interfaces estables. Desacopla las interfaces estables de las variaciones requeridas por las instanciaciones de una aplicación, en un contexto particular.

Inversión del control el propio *framework* determina el conjunto de métodos a aplicar en respuesta a eventos externos, produciéndose el control por parte del *framework*.

Desventajas

Esfuerzo de desarrollo el desarrollo de un *framework* reutilizable requiere un mayor esfuerzo y experiencia que el desarrollo de aplicaciones convencionales.

Curva de aprendizaje el tiempo necesario para aprender a desarrollar productos, a partir de un *framework*, es inicialmente más alto que el utilizado en el aprendizaje para desarrollar una aplicación individual. Este esfuerzo se ve recompensado posteriormente, porque la utilización del *framework* para producir nuevos productos reduce los tiempos necesarios, acelerando el desarrollo.

Integrabilidad la integración entre *frameworks* es un punto problemático. Los *frameworks* están diseñados desde un principio para su instanciación, no así para su integración con otros.

Facilidad de mantenimiento los cambios en los requisitos también afectan a los *frameworks*, y los cambios que se producen sobre ellos, también afectan a las aplicaciones que los utilizan. Esto provoca que tanto el *framework* como las correspondientes aplicaciones que los usan evolucionen a la par.

Validación y depuración el alto grado de abstracción del *framework* dificulta su prueba, que tiene que realizarse siempre en el contexto de una instanciación concreta. Además se añade la dificultad de distinguir entre errores en el *framework* o en el código de aplicación. La inversión de control añade el problema adicional de seguir la lógica de control, que salta entre ambas partes.

Eficiencia el uso de niveles adicionales de indirección afecta a la eficiencia de la aplicación. Aunque esto no es relevante en muchas aplicaciones.

Carencia de estándares en la actualidad no existen estándares para el diseño, implementación, documentación y adaptación de *frameworks*.

Comparando las ventajas y desventajas que aportan, se puede concluir que aunque exigen un mayor esfuerzo de desarrollo inicial por parte del diseñador y programador, desde el punto de vista del usuario final, las ventajas obtenidas justifican su uso, salvo por una cierta dificultad añadida en su aprendizaje inicial que en siguientes desarrollos se va reduciendo.

6.1.3. Clasificación de *frameworks*

Aunque los *frameworks* pueden ser clasificados desde diferentes aspectos, la siguiente clasificación toma como criterio la técnica usada para extenderlos:

Caja blanca (*white box*) utilizan intensivamente mecanismos propios de la orientación a objetos, como la herencia y la ligadura dinámica. La funcionalidad existente es reutilizada y extendida por herencia de las clases bases y redefinición de los métodos de enganche. El desarrollador necesita tener un profundo conocimiento de la jerarquía de herencia del *framework*.

Caja negra (*black box*) soportan la extensión definiendo interfaces para componentes que pueden ser conectados a través de la composición. La funcionalidad existente es reutilizada definiendo componentes que conforman con la interfaz, e integrando los componentes. Son generalmente más fáciles de usar que los de caja blanca, aunque son más difíciles de desarrollar porque requieren un alto número de interfaces que anticipen necesidades futuras.

Caja gris (*grey box*) evitan los problemas de los *frameworks* de caja blanca y negra, dotando de suficiente flexibilidad y extensibilidad, ocultando información innecesaria al desarrollador de aplicaciones.

Como se indica en [Fayad et al., 1999], inicialmente se desarrolla un *framework* de caja blanca, para una vez que se ha entendido suficientemente, transformarlo en un *framework* de caja negra. Esta evolución es habitual que se produzca, aunque normalmente el *framework* realmente llega a ser un *framework* de caja gris evolucionado a partir de uno de caja blanca [Markiewicz and de Lucena, 2001].

Desde ese punto de vista, se puede establecer que los *frameworks* de caja blanca se instancian sólo a través del desarrollo de nuevas clases. Por otro lado, los de caja negra y gris producen instancias a través de la escritura de guiones (*scripts*) de configuración y el uso de herramientas automáticas que creen clases y nuevo código fuente. Como se detallará en la Sec. 6.3, es importante conocer dicha clasificación, dado que la propia evolución del *framework* que se propone en este trabajo ha seguido paso a paso esta evolución.

6.1.4. Documentación de *frameworks*

La capacidad de crear sistemas ejecutables a partir de un *framework* viene determinada por la utilidad de sus mecanismos de instanciación, pero también en gran medida de la documentación asociada.

La gran curva de aprendizaje del uso de un *framework* es un impedimento para su uso. Para evitar este problema, la documentación del *framework* debe ser adecuada para una fácil reutilización por parte de sus usuarios. La documentación que se aporta en las siguientes secciones persigue facilitar dos tareas básicas:

Mantenimiento del *framework* desde el punto de vista de la persona responsable de mantener y evolucionar el *framework*, debe comprender y conocer el diseño del *framework*, así como el dominio de aplicación y la flexibilidad requerida. Otros aspectos como el dominio, arquitectura general, motivaciones para la elección de los *hot spots*, así como los patrones de diseño utilizados en los *frozen spots* y su motivación, serán detallados a continuación.

Desarrollo de aplicaciones desde el punto de vista de la persona que crea nuevas aplicaciones utilizando el *framework*, se desea saber cómo personalizar el *framework* para producir la nueva aplicación. La prioridad es explicar cómo se llega al objetivo, sin detallar por qué se trabaja de ese modo. Se debe tener en cuenta que el usuario no es un experto en el dominio, y puede no ser un desarrollador muy experimentado.

De todos los elementos habituales para documentar un *framework* se seguirán en mayor o menor medida los siguientes: visión general del *framework*, patrones de diseño, recetas y libros de cocina [Fayad et al., 1999].

Mientras que en esta sección se detallará la documentación desde el punto de vista de la persona responsable de mantener el *framework*, en el Capítulo 7 se detallará una instanciación concreta para construir y ejecutar refactorizaciones sobre un lenguaje objetivo real como Java.

6.1.5. *Frameworks* en refactorización

Se ha planteado brevemente el concepto de *framework*, sus beneficios, problemas, clasificaciones y recomendaciones respecto a su documentación. Llegado a este punto se debe justificar

el por qué se escoge como solución de soporte a la refactorización con cierto grado de independencia del lenguaje.

Como se ha indicado previamente, la construcción de aplicaciones basadas en *frameworks* no es simple (ver el Apartado 6.1.2), sin embargo, desde un punto de vista de la reutilización, presenta un gran número de ventajas. La capacidad de reutilización del código y del diseño de *frameworks* orientados al objeto permite una productividad mayor y un tiempo breve en el desarrollo de aplicaciones, a posteriori, en comparación con el desarrollo tradicional de los sistemas de software, orientados a la reutilización de bibliotecas o APIs.

Si como resultado del presente trabajo, no sólo se aporta una sólida base para el desarrollo y evolución de herramientas en refactorización sobre varios lenguajes, sino que se aporta un conjunto de *frameworks* instanciables y extensibles para otros tipos de tareas, el esfuerzo realizado se verá recompensado en posteriores etapas.

Para nuestro problema concreto, tenemos por un lado el uso de *frameworks* para modelar los puntos fijos o congelados, respecto a los conceptos identificados de refactorización y del metamodelo MOON. Por otro lado, se permite la introducción de puntos calientes a ser implementados en las instanciaciones para diferentes lenguajes y en las implementaciones concretas de las refactorizaciones. Se da con esto soporte a la refactorización para un amplio conjunto del lenguajes objetivo, tal y como se estableció en los objetivos iniciales.

6.2. Representación del código

Las herramientas de refactorización necesitan manejar la información presente en el código fuente, dado que posteriormente se necesita consultar y modificar dicha información incluida para llevar a cabo las refactorizaciones. El problema principal es que la representación elegida permita realizar estas operaciones de la forma más simple y efectiva posible.

Actualmente, la mayoría de soluciones están basadas en árboles de sintaxis abstracta (*Abstract Syntax Tree* o AST) y patrones de diseño bien conocidos como **Visitor** [Gamma et al., 1995]. El principal problema derivado del uso del AST, es la atadura a un único lenguaje, o más concretamente a su gramática. Esto es un gran obstáculo para la reutilización de las refactorizaciones implementadas para un lenguaje objetivo concreto.

Otras líneas de trabajo, apuntan a gestionar la información del código a través de bases de datos relacionales. Las consultas son generadas gracias a la potencia del lenguaje de consulta SQL. Aunque relativamente fácil de adoptar, esta solución permite la detección de oportunidades de refactorización y un análisis de su aplicabilidad, pero impide la transformación sobre el modelo relacional utilizado, por no disponer de toda la información inicial del código.

La utilización de soluciones basadas en lógica de predicados y lenguajes lógicos [Bravo, 2003, Kniesel and Koch, 2004] presenta las mismas fortalezas y debilidades. El problema surge a la hora de regenerar o recuperar el código a partir de la información almacenada, que suele ser un subconjunto de la información inicial, por lo que una regeneración del código completa es inviable.

Nuestra propuesta se basa en el empleo del metamodelo MOON como base inicial y su especialización para cada lenguaje. Por un lado, permite la consulta de la información almacenada, con capacidades similares a las ya mencionadas. Por otro lado, permite cambiar el estado actual del modelo de objetos, y obtener el código refactorizado sin pérdida de información.

Adicionalmente, surgen nuevas oportunidades de reutilización partiendo de los beneficios de una solución basada en *frameworks*. Un ejemplo de esto es la recolección de métricas de código con independencia del lenguaje [Crespo et al., 2005a, Crespo et al., 2005b, Crespo et al., 2006a], manejando los puntos comunes y los puntos de variación para cada lenguaje.

Aunque el metamodelo MOON puede representar puntos comunes y variantes generales, no incluye todas las características de los lenguajes de programación. Así pues, con el objetivo de resolver las debilidades presentes en otras soluciones, es necesario dar soporte a la variabilidad a través de puntos de extensión para características particulares.

Una simple observación de características bien conocidas de lenguajes de la corriente principal, permiten que podamos enumerar algunos casos como por ejemplo:

- una instanciación del *framework* para JAVA debe incluir conceptos como excepciones, interfaces, bloques `try-catch-finally`, anotaciones, etc.
- una instanciación del *framework* para EIFFEL debe incluir conceptos de anclas, diseño por contrato, agentes, etc.
- una instanciación del *framework par*a .NET debe incluir conceptos como delegados, eventos, propiedades, etc.

En este sentido, se establece una taxonomía de conceptos en base a su inclusión en el metamodelo y su extensión:

1. **General**: son elementos contenidos en la mayoría de lenguajes orientados a objetos estáticamente tipados *e.g.* clase, método o atributo, etc. incluidos en la mayoría de lenguajes, como EIFFEL, C++, JAVA, C#, etc. Las características comunes son representadas en MOON como clases del metamodelo y reutilizadas sobre varios lenguajes.
2. **Extensible**: incluidas en la mayoría de lenguajes pero en cada uno se da una semántica diferente. Por ejemplo, los modificadores de acceso, reglas de herencia, etc. como conceptos están presentes en la mayoría de lenguajes, pero de diferentes formas. Estos se modelan como métodos abstractos en MOON que deben ser implementados de manera concreta en la instanciación del *framework*, permitiendo trabajar a partir de las abstracciones.
3. **Particular**: los conceptos del lenguaje objetivo que no están incluidos en el núcleo de MOON. La instanciación para el lenguaje concreto incluye estos conceptos para soportar el conjunto completo de características del lenguaje, tanto para su consulta, como para su modificación y regeneración.

Al analizar un código escrito en un LPOO, estáticamente tipado, se produce un proceso de instanciación sobre las abstracciones del lenguaje definidas. Debido a que el metamodelo MOON no representa todas las características de las familias de lenguajes a tratar, es necesario permitir la instanciación de las nuevas características para lenguajes concretos. Esto se confirma en la decisión de diseño tomada en el núcleo de MOON, en el que todas las clases son abstractas, entendiendo que nunca se llegan a producir instancias directas, sino instancias a través de las clases que las extienden sobre un lenguaje concreto. En [López et al., 2003] se mostró un primer estudio de una extensión de Java para la definición de refactorizaciones, que ha sido modificada en posteriores trabajos.

Una vez disponible la extensión del lenguaje concreto, el proceso es similar al de otras soluciones descritas previamente: se toma el código fuente y bibliotecas de las que depende, y se almacena su información sobre instancias del modelo, como se muestra en la Fig. 6.1. Más concretamente se tienen instancias del lenguaje particular ($L_1..MOON$ o $L_2...MOON$), constituyendo un grafo que modela el código inicial, pero además al extender las abstracciones de MOON, permite que se pueda trabajar a un nivel superior sobre las clases de MOON.

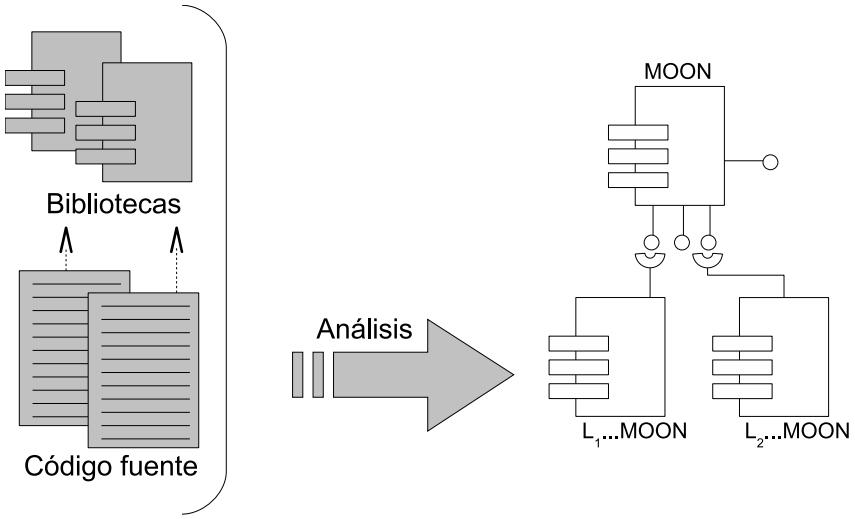

Figura 6.1: Fase de análisis de código y generación del modelo

En este proceso de análisis es fundamental, y no siempre ha sido contemplado en este tipo de soluciones, una carga de información de las bibliotecas de clases, contenidas físicamente en ficheros binarios. Aunque dichos elementos no sean objetivo de las refactorizaciones, puesto que el objetivo final es únicamente refactorizar código fuente, su información es indispensable para un análisis correcto del código, y su posterior transformación.

Como ejemplo, se pueden tomar lenguajes como Eiffel, Java, o lenguajes de la familia .NET, que se basan en el concepto de una jerarquía de tipos única con un elemento raíz universal. Las propiedades de dicha clase están disponibles en todas las clases construidas (puesto que directa o transitivamente toda clase hereda de ella). La información de este tipo de clases, así como las contenidas en bibliotecas de utilidades o acceso al sistema de ficheros,

no están disponibles inicialmente en el código fuente, e incluso en tal caso, no serían objetivo de refactorización[2]. Por lo tanto se deben analizar y extraer dicha información a partir de las bibliotecas binarias disponibles (*e.g.* ficheros `.jar`, `.exe`, `.dll`, etc). Así pues, el modelo recoge de manera completa la información que se puede extraer del código y de las bibliotecas de las que éste dependiera, para completar el proceso.

6.3. Construcción y ejecución de refactorizaciones

Establecidos los elementos básicos que componen las refactorizaciones en el Capítulo 4, queda por fijar la estructura del *framework* que pone en movimiento todas las piezas descritas para su aplicación práctica. En [Crespo et al., 2004], se planteó un diseño de motor de refactorizaciones que permitía ejecutar refactorizaciones definidas sobre las precondiciones, acciones y postcondiciones previamente construidas.

La principal fortaleza de esta solución es que está planteada para ser extendida y utilizada con independencia del contexto o modelo subyacente. De hecho en posteriores trabajos, se planteó su uso no sólo con el metamodelo MOON para lenguajes de programación, sino para modelos UML [López et al., 2006b], en el contexto del diseño software, mediante el uso del patrón **Adapter** [Gamma et al., 1995]. El motor permite incluir el concepto de refactorización de manera natural e independiente del objetivo final sobre el que se aplique.

En los siguientes apartados se detallan los elementos constituyentes del *framework*, describiendo la evolución que éste ha sufrido desde el punto de vista de un *framework* de caja blanca hacia un *framework* de caja gris.

El motor de refactorizaciones ejecuta las refactorizaciones sobre un modelo de objetos inicial y obtiene un nuevo modelo de objetos ya refactorizado. Las refactorizaciones deben ser definidas para facilitar por un lado su ejecución y por otro, la reutilización e instanciación en la construcción de nuevas refactorizaciones. En esta sección se muestra la definición y diseño del núcleo del motor de refactorizaciones así como una extensión de los elementos, operando sobre el metamodelo MOON. El *framework* ha sido definido para permitir su reutilización, como se puede ver en la Fig. 6.2.

6.3.1. Núcleo del motor de refactorizaciones

Partiendo de lo establecido en el Capítulo 4 en su Apartado 4.1.2, se definió una plantilla con una serie de elementos descriptivos (nombre, descripción y motivación), argumentos de entrada, precondiciones, acciones y postcondiciones, pasos que forman un algoritmo común a todas las refactorizaciones. En concreto, los elementos básicos que participan en la ejecución de una refactorización son:

[2]Una modificación y regeneración de bibliotecas del núcleo del lenguaje, sería sólo válida para el producto particular, pero no en otros contextos, derivando en problemas graves posteriormente.

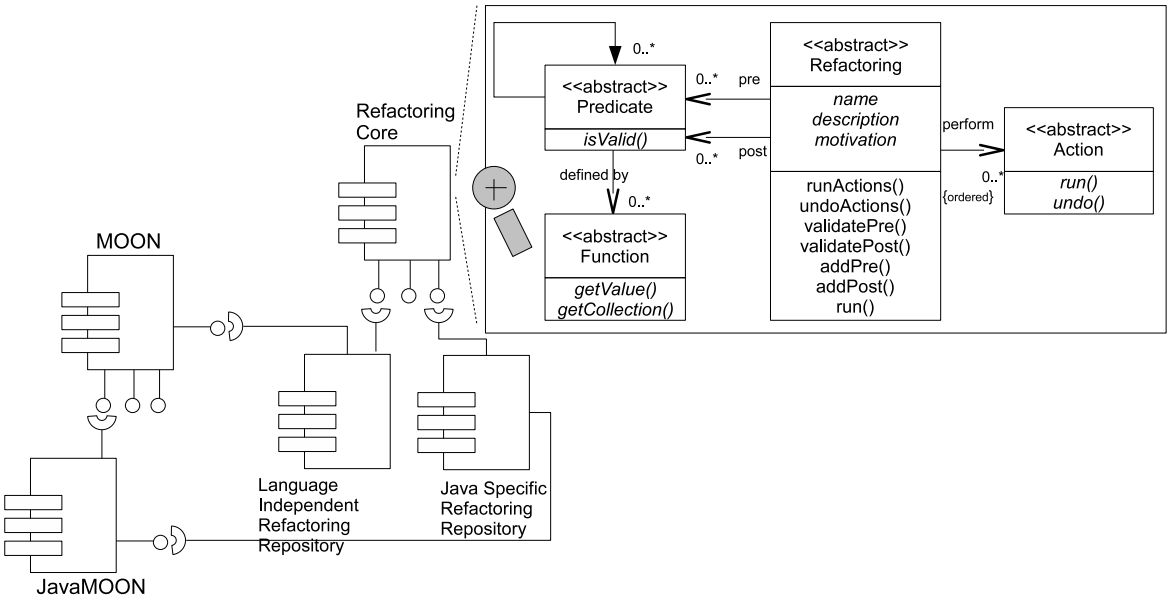

Figura 6.2: Motor de refactorizaciones

Entradas elementos de inicio que marcan el punto de arranque de la refactorización. El resto de elementos utilizan la información proporcionada directamente o bien calculada a partir de dichas entradas.

Precondiciones se identifica una fase de comprobación de predicados que tienen que verificarse para poder llevar a cabo la refactorización. Esto permite garantizar en una cierta medida la preservación del comportamiento.

Acciones operan directamente sobre las instancias que representan los elementos, modificando el estado actual del código obtenido.

Postcondiciones se identifica una fase de comprobación de predicados que deben verificarse a la finalización de las acciones. Si alguna de las acciones causó errores, reflejados en el lanzamiento de excepciones, se requiere realizar una serie de acciones que deshagan los efectos para dejar las instancias en un estado consistente.

El diseño de las clases que forman el núcleo del motor, y operan sobre los elementos anteriormente descritos, se representa en la Fig. 6.2. El algoritmo de ejecución de refactorizaciones es común a todas ellas dejando como puntos de enganche la extensión de elementos que participan en una refactorización particular.

El patrón de diseño **Template Method** [Gamma et al., 1995] se aplica en la clase `Refactoring`. El método `run` cumple el papel de *método plantilla* invirtiendo el control

de flujo [Johnson and Foote, 1988] o también denominado *"principio de Hollywood"* [Sweet, 1985]. Con este esquema, una refactorización puede ser vista como piezas implementadas con clases de un repositorio: predicados y funciones (componiendo las pre/postcondiciones), y por otro lado las acciones (realizando las transformaciones sobre el código). El rol de comando (**Command** [Gamma et al., 1995]) es tomado por las clases `Action`. Una acción puede deshacerse o quedar registrada cuando una excepción ha sido lanzada en el proceso de ejecución. `Predicate` y `Function` son también variaciones de **Commmand**.

Cada refactorización es implementada como una extensión de una clase `Refactoring`. Las clases abstractas como `Predicate`, `Function` y `Action` deben ser extendidas por clases concretas en los repositorios. Los constructores de las refactorizaciones toman las entradas y ensamblan las precondiciones, acciones y postcondiciones en el orden indicado. Los puntos de variabilidad se implementan también a través de los constructores particulares de cada clase.

El *método plantilla* se presenta en el Código 6.3.1[3]. La clase se diseña como abstracta puesto que no se requiere instanciar directamente.

Código 6.3.1: Método plantilla en la clase `Refactoring`

```
public void run() throws PreconditionException,
                         PostconditionException {
    try{
        validatePreconditions();
        runActions();
        validatePostconditions();
    }
    catch (PostconditionException postconditionEx) {
        // Policy of organized panic.
        undoActions();
        throw postconditionEx;
    } // catch
} // run (Template Method PD)
```

En este punto queda abierto para introducir en futuras versiones la utilización del patrón **Memento** [Gamma et al., 1995] para guardar el estado antes de realizar las acciones y facilitar las operaciones de deshacer.

Para construir nuevas refactorizaciones, es necesario definir instanciaciones concretas a los elementos del núcleo. Para ello se definen cada una de las refactorizaciones concretas como por ejemplo **Add Class**, **Remove Method** [Fowler, 1999], **Parametrizar**, etc., como instanciaciones concretas de la clase `Refactoring`.

Si las pre/postcondiciones o acciones sólo usan conceptos generales o extensibles de MOON, son clasificadas como independientes del lenguaje. Por otro lado, si usan características particulares del lenguaje, son clasificadas como dependientes del lenguaje. Dependiendo de su clasificación los elementos residen en sus correspondientes repositorios.

Esto permite reutilizar las mismas consultas o transformaciones cuando se implementan refactorizaciones para distintos lenguajes, siempre y cuando sean catalogados como indepen-

[3]Se utiliza Java como lenguaje de implementación para el *framework*.

dientes del lenguaje o extensibles (sólo tienen dependencias del metamodelo MOON y no de una extensión concreta de un lenguaje).

Por ejemplo la precondición **`ExistParameterWithSameName`** o la acción **`MoveAttributeAction`**, almacenadas en el repositorio, son reutilizables para varios lenguajes, puesto que para la totalidad de lenguajes objeto de estudio, se observa el concepto abstracto de que todo parámetro formal tiene un nombre y las clases contienen atributos.

En casos ideales, una operación de refactorización podría ser reutilizada como un todo para varios lenguajes, pero en la mayoría de los casos, los elementos de una refactorización son reutilizados y adecuadamente compuestos introduciendo variaciones y partes dependientes del lenguaje.

6.3.2. Construcción y ejecución estática

Para la construcción de una refactorización, entendiendo que se produce una instanciación concreta de una refactorización del *framework* se siguen los siguientes pasos:

1. Definir los elementos de la refactorización (entradas, pre/postcondiciones y acciones).
2. Revisar que los elementos existen en los repositorios, identificando si son independientes o dependientes del lenguaje (esto marca a qué repositorio concreto se debe acudir). Si los elementos no existen deben ser implementados e incluidos en el repositorio.
3. Construir una clase que herede de `Refactoring` e incluirla en el correspondiente repositorio (paquete).
   - en el constructor añadir las entradas de la refactorización
   - añadir los atributos privados correspondientes a las entradas
   - instanciar las precondiciones (opcional) y añadirlas a la refactorización (método `addPrecondition`)
   - instanciar las acciones y añadirlas a la refactorización (método `addAction`)
   - instanciar las postcondiciones (opcional) y añadirlas a la refactorización (método `addPostcondition`)
4. Compilar la refactorización.

Una segunda fase se produce cuando se quiere ejecutar la refactorización construida. Dependiendo del tipo de interfaz que se ofrezca al usuario, se seguirán los siguientes pasos:

1. Seleccionar el elemento a refactorizar.
2. Seleccionar la refactorización a aplicar.
3. Introducir las entradas adicionales.

4. Ejecutar la refactorización en el motor.

El motor carga la refactorización e inicia el proceso de aplicación, transformando la representación actual del código que se tiene en memoria: se validan las precondiciones, se aplican las acciones y finalmente se comprueban las postcondiciones. En caso de error el *framework* da soporte a las operaciones deshacer y registro de *log*.

Esta solución es válida en tanto en cuanto las refactorizaciones no estén sujetas a continua revisión y modificación, sin embargo, en la práctica, es habitual que las refactorizaciones estén en un proceso iterativo de análisis, definición y construcción (ver la Sec. 4.3). Toda modificación tiene un efecto de modificación del código fuente de la refactorización y recompilación, que puede afectar a una utilización práctica del motor como *framework* de caja blanca.

6.3.3. Evolución hacia *framework* de caja negra

La mayoría de las herramientas de refactorizaciones implementan sus operaciones como composiciones prefijadas, codificadas en el lenguaje de implementación concreto. Aunque a lo largo del tiempo se ha tendido a mejorar el diseño [Frenzel, 2006, JetBrains, 2013a], las refactorizaciones son difíciles de cambiar, mantener y reutilizar.

Una vista simplificada de las refactorizaciones como la composición de consultas y transformaciones conduce a pensar en un modo más composicional, usando elementos en un determinado orden, y añadiendo información adicional.

Si se es capaz de gestionar las partes de las refactorizaciones como piezas aisladas, éstas pueden ser ensambladas en tiempo de compilación o ejecución. Con este propósito, se evolucionó la propuesta inicial, de construir y ejecutar refactorizaciones en módulos codificados a mano como se ha detallado en la Apartado 6.3.2, hacia un ensamblaje dinámico, como se puede ver en la Fig. 6.3.

Los elementos de las refactorizaciones no son directamente referenciados en el código, sino en ficheros XML. Por ejemplo, una refactorización como **Rename Class** tendría su contrapartida en un fichero `RenameClass.xml`

El motor de refactorizaciones presenta una variante para cargar los ficheros con las descripciones de las refactorizaciones bajo demanda. Las precondiciones, acciones y postcondiciones, son también analizadas desde la definición previa en XML. Las partes son extraídas y ensambladas desde los repositorios. La refactorización es construida en tiempo de ejecución, reutilizando el motor ya presentado en Apartado 6.3.1. La refactorización está preparada para ser invocada para un conjunto de entradas (tomadas como instancias de un modelo), obteniendo las instancias modificadas.

La interfaz necesaria para introducir las entradas de la refactorización puede ser generada también en tiempo de ejecución, consultando al usuario los valores a introducir. En soluciones previas, cada refactorización necesita ser personalizada, creando su propia ventana (codificada manualmente o con ayuda de editores visuales), mientras que la solución dinámica resuelve

el problema en un único paso, con la desventaja de no poder personalizar el aspecto de la representación gráfica.

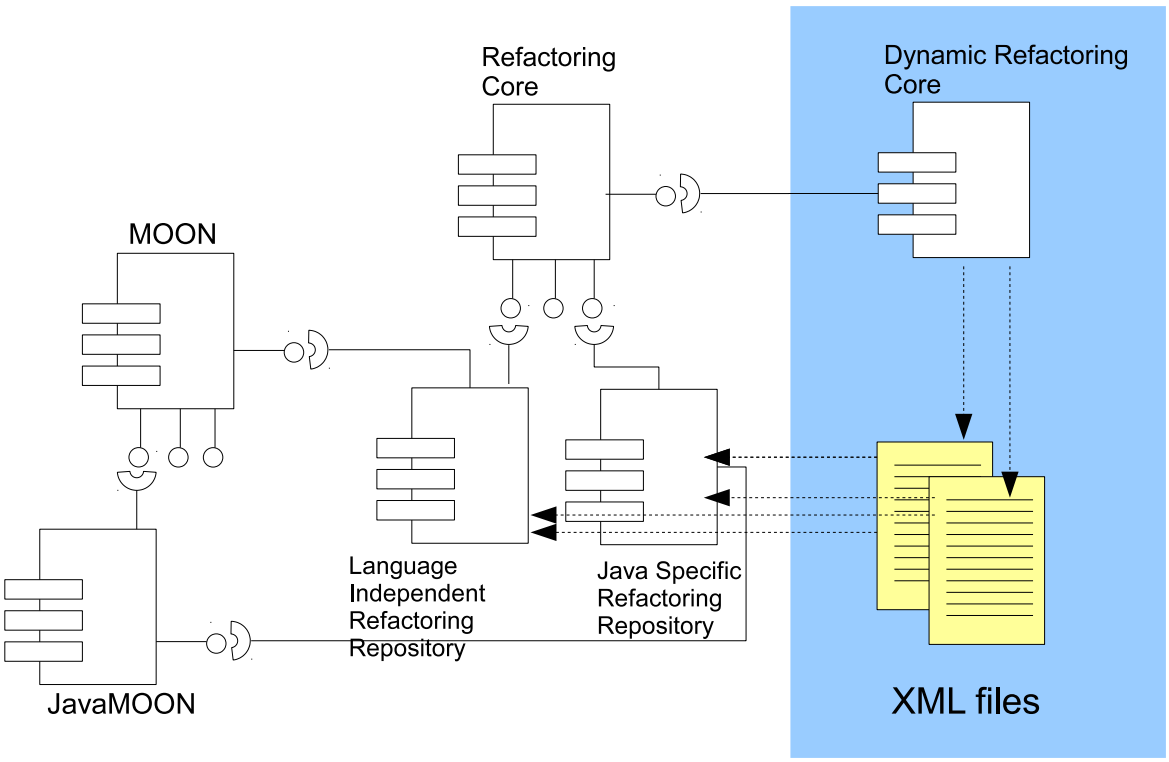

Figura 6.3: Ejecución dinámica de refactorizaciones

6.3.4. Construcción y ejecución dinámica basada en XML

La aproximación dinámica necesita un proceso que muestra ciertas diferencias respecto a un desarrollo estático. Para incluir una nueva refactorización, se establecen los siguientes pasos:

1. Definir los elementos de una refactorización.
2. Revisar que los elementos existen en los repositorios, identificando si son independientes o dependientes del lenguaje (esto marca a qué repositorio concreto se debe acudir). Si los elementos no existen deben ser implementados e incluidos en el repositorio.
3. Construir el correspondiente fichero XML y almacenarlo.
   - incluir información general
   - seleccionar las entradas

- seleccionar las precondiciones (opcional)
- seleccionar las acciones
- seleccionar las postcondiciones (opcional)
- introducir códigos de ejemplo (opcional)

Cuando el usuario/desarrollador aplica la refactorización construida, de acuerdo a la interfaz generada, se seguirán los siguientes pasos:

1. Seleccionar el elemento a refactorizar.
2. Seleccionar la refactorización a aplicar.
3. Introducir las entradas adicionales.
4. Ejecutar la refactorización en el motor.

Como se puede observar, la segunda fase no se diferencia de la solución estática de instanciación del *framework* (ver Apartado 6.3.2). Esto se debe a la reutilización completa del motor en ambas soluciones, simplificándose la construcción de las refactorizaciones por parte del usuario del *framework*.

6.3.5. Elementos XML de una refactorización

Las refactorizaciones han sido definidas como un conjunto de elementos: entradas, pre/post-condiciones y acciones. Estos elementos tienen sus diferentes enlaces con el fichero XML (ver Fig. 3.11). Las entradas son identificadas como elementos del metamodelo MOON o de la extensión concreta, con un atributo de tipo, que indica claramente si es independiente (`moon.core`) o dependiente del lenguaje (*`xxx`*`moon.core`, donde *`xxx`* se corresponde con el nombre del lenguaje objetivo).

Un nombre califica al elemento, para ser referenciado en otros elementos (pre/post y acciones). Se utilizan entradas especiales como guía para la construcción de interfaces gráficas, como por ejemplo el elemento raíz que dirige la refactorización.

Cuando las refactorizaciones necesitan entradas derivadas de otras entradas, se incluye un atributo `from`. Finalmente, el mecanismo de reflexión encontrará el método con el tipo de retorno adecuado:

```
<input name="Class" root="true" type="moon.core.classdef.ClassDef"/>
<input from="Class" method="getName"
    name="OldName" root="false" type="moon.core.Name"/>
<input name="NewName" root="false" type="moon.core.Name"/>
```

Las precondiciones y postcondiciones referencian predicados para ser evaluados. Estos elementos son localizados en los repositorios (MOON y particular del lenguaje). Si no existen,

un subproceso de definición e implementación debe ser completado antes de continuar con la construcción de la refactorización. Los parámetros de las funciones necesarias son añadidos usando los nombres de las entradas previamente indicadas, y es obligatorio que todos los argumentos sean entradas o valores derivados de las entradas:

```
<precondition
  name="repository.moon.concretepredicate.NotExistsClassWithName">
<param name="NewName"/>
</precondition>
```

Las acciones usan la misma estrategia, incluyendo los elementos objetivo a ser transformados:

```
<action
  name="repository.moon.concreteaction.RenameClass">
<param name="Class"/>
<param name="NewName"/>
</action>
```

Opcionalmente, las postcondiciones son incluidas para comprobar la correcta ejecución:

```
<postcondition
   name="repository.moon.concretepredicate.NotExistsClassWithName">
<param name="OldName"/>
</postcondition>
```

Siempre se considera el modelo de objetos (grafo actual) como entrada implícita. La salida por defecto es el nuevo grafo refactorizado, y no es explícitamente nombrado en el fichero XML. El fichero XML es validado contra una declaración de tipo de documento (DTD) para comprobar su corrección.

6.3.6. Asistentes en la construcción de refactorizaciones y uso de reflexión

Debido a que la escritura de XML de forma manual es propensa a errores, se debe proporcionar una interfaz gráfica que asista a su construcción. El asistente o *wizard* guía en la construcción correcta, a través de varios pasos:

1. introducir información general de la refactorización.
2. seleccionar las entradas.
3. añadir pre / postcondiciones y acciones.
4. incluir códigos de ejemplo.
5. confirmar cambios.

Los elementos referenciados de los repositorios son validados usando un mecanismo de reflexión. Se reutilizan los elementos binarios previamente creados. Si las clases no pueden ser cargadas, no son mostradas en las ventanas del asistente. Así pues, la definición final asegura utilizar sólo elementos ejecutables existentes y que puedan ser invocados con seguridad.

Una vez completada la refactorización, el fichero XML debe ser salvado en una localización específica, donde el motor, en su variante de carga dinámica, pueda encontrarlo y analizarlo (ver Fig. 3.11).

La herramienta dinámica detecta las refactorizaciones disponibles, leyendo la información y mostrando el conjunto de refactorizaciones aplicables dependiendo del elemento seleccionado en el modelo (elemento raíz en el fichero XML): *e.g.* paquetes, clases, métodos, etc.

Cuando el usuario quiere ejecutar una refactorización debe seleccionarla, creándose una ventana dinámicamente. Los campos a introducir se deducen de las entradas en el fichero XML, marcando el esquema de la pantalla a visualizar. El usuario debe introducir nuevo valores para cada uno de los campos.

Después de introducir los puntos de entrada, las precondiciones, acciones y postcondiciones son extraídos, recuperados de los repositorios y parametrizados a partir de las entradas. Usando mecanismos de reflexión, el motor dinámico resuelve el conjunto de clases (usando su nombre completo cualificado), construye la refactorización (mediante la carga dinámica de clases) e introduce la refactorización en el motor para su ejecución (ver Fig. 6.2 y Fig. 3.11). Las refactorizaciones se aplican y provocan cambios en el modelo actual. Finalmente, el código refactorizado debe ser regenerado desde el nuevo estado del modelo de objetos que representa el código fuente.

6.4. Regeneración del código refactorizado

La regeneración de código en el lenguaje origen pasa por generar el código fuente a partir de la información almacenada como instancias de los metamodelos (finalmente como instanciaciones del *framework*).

Puesto que el proceso de refactorización exige que el código refactorizado preserve su comportamiento inicial, se debe recuperar el código de tal forma que pueda volverse a generar (compilar) una representación binaria ejecutable que pueda ser probada de nuevo.

Este proceso de regeneración o recuperación del código, completa el proceso de ida y vuelta (*roundtrip*). En otro tipo de contextos, como en el análisis estático o dinámico del código, el cálculo de métricas, la visualización o representación gráfica del código, etc, no suele tenerse en cuenta este problema.

Esta sección se desglosa en dos apartados: la primera describe la problemática asociada a la regeneración de código en base a la solución propuesta, y en segundo lugar se describe una solución de diseño para el módulo de regeneración de código, a integrar en la solución basada en *frameworks* descrita en anteriores secciones.

6.4.1. Regeneración a partir de representaciones intermedias

El metamodelo MOON es una representación intermedia donde se recogen las abstracciones comunes, sin embargo, no recoge de manera completa los elementos que forman el código fuente o binario de los componentes físicos de un programa (fichero fuente o binario, posición - línea/columna en el código, etc.) La transformación hacia MOON puede llevar asociada una pérdida de información que impide una regeneración directa del código fuente en el lenguaje origen, ya sea éste refactorizado o no.

El problema planteado es decidir cómo y dónde almacenar la información, sin que exista pérdida en la regeneración. En particular el problema aumenta en lo referente a los cuerpos de los métodos, donde el conjunto de instrucciones y expresiones suele ser bastante complejo, no manteniendo muchas partes comunes entre diferentes lenguajes.

En trabajos previos [López and Crespo, 2003], se plantearon diversas soluciones para evitar la pérdida de información, siendo este un problema que se puede abordar de diferentes formas:

- Almacenar de forma textual el cuerpo completo de los métodos en su lenguaje original, ligado a las instancias del metamodelo. Esta propuesta no deja cerrada la posibilidad de modificar el cuerpo de los métodos, ya que se pueden realizar tratamientos textuales sobre el código fuente original como se realiza en los frontales (*front-end*) de MOOSE [Tichelaar, 2001].

- Almacenar sobre las distintas instanciaciones el AST correspondiente a la definición del cuerpo de los métodos. Las modificaciones se llevarían a cabo modificando el AST, particularizando para cada lenguaje.

- Almacenar en las instancias punteros al fichero fuente inicial modificando directamente el texto del fichero a la hora de refactorizar. Esta solución ha sido planteada en [Strein et al., 2006].

Aunque estas soluciones son aplicables, presentan ciertos problemas. Las consultas y modificaciones al código en una representación de texto plano adolece de ciertas garantías en la corrección de la aplicación de refactorizaciones. La comprobación de condiciones basada en texto no puede aprovecharse de la información adicional del metamodelo (tipos, reglas, etc.), pudiendo no garantizar una correcta aplicación de la refactorización.

Por otro lado, las modificaciones sobre un AST están íntimamente ligadas a la gramática subyacente, limitando un enfoque que potencia la reutilización. Además, de manera general se produce una cierta impedancia al escoger una solución híbrida en la que cierta información se almacena como instancias de un metamodelo y otra se almacena como texto o AST.

Para evitar este tipo de problemas a la hora de almacenar y modificar la información del código, se ha planteado como solución la definición de instanciaciones concretas del *framework* para cada lenguaje, en donde sí se almacena de manera completa la información necesaria del código para permitir una recuperación a partir del mismo.

Aunque en su contra se puede indicar que el número de clases definidas para almacenar todo el código puede alcanzar un tamaño medio o grande, también es cierto que este problema aparece en todas las soluciones actualmente implementadas en las herramientas comerciales para cada lenguaje concreto.

Así pues, toda instancia del *framework* para el lenguaje concreto L_i, almacenará información sobre su posición en el código fuente, si éste está disponible, de tal forma que a partir de dichas instancias se pueda volver a reconstruir el código inicial o transformado tal y como se describe en el Capítulo 7.

6.4.2. Módulo de regeneración de código

Un regenerador de código obtendrá la información a partir de las instanciaciones específicas del lenguaje, que a través de herencia han obtenido la funcionalidad básica definida en el metamodelo MOON.

El regenerador recorre las instancias de la instanciación concreta, utilizando el patrón de diseño **Visitor** [Gamma et al., 1995], pidiendo a cada elemento la obtención de su código asociado. De esta forma se obtienen de nuevo los códigos fuente (ver Fig. 6.4) después de ser transformados.

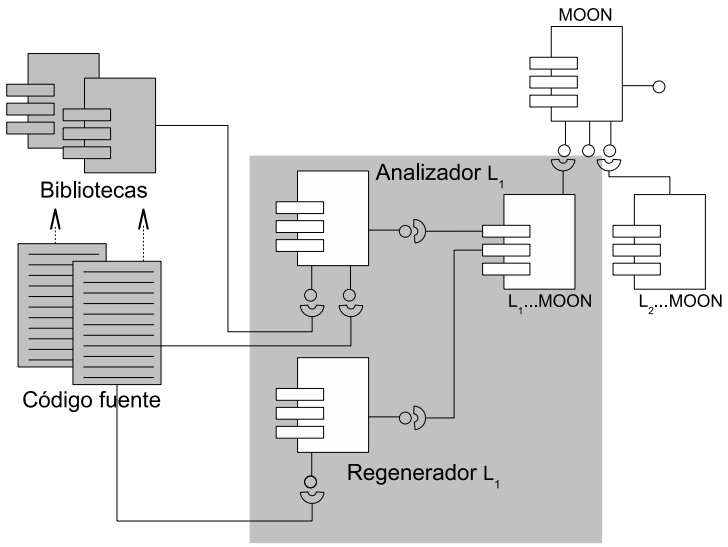

Figura 6.4: Regeneración del código

La interfaz `Visitor` se establece desde un punto de vista independiente del lenguaje, definiéndose sobre elementos del metamodelo de MOON (ver Código 6.4.2) pudiendo, en la implementación concreta del lenguaje, implementar de manera adecuada estos métodos:

Código 6.4.2: Interfaz `Visitor`

```
public interface Visitor {
    void visitNameSpace(NameSpace ns);
    void visitClassDef(ClassDef cd);
    void visitClassType(ClassType ct);
    void visitMethod(Method md);
    void visitAttribute(Attribute ad);
    void visitInheritanceClause(InheritanceClause ic);
}
```

En la Fig. 6.5 se muestra el diseño básico del módulo de regeneración. La interfaz `Visitor` recopila la información del modelo (`Model`) y debe completar el código de un objeto que implemente la interfaz `SourceCode`. Se habilitan distintos tipos de estrategias de recuperación siguiendo el patrón **Strategy** (implementado como `Strategy` y `RegenerationStrategy`) que coordinan la navegación, para que posteriormente se pueda recuperar la estructura física correspondiente.

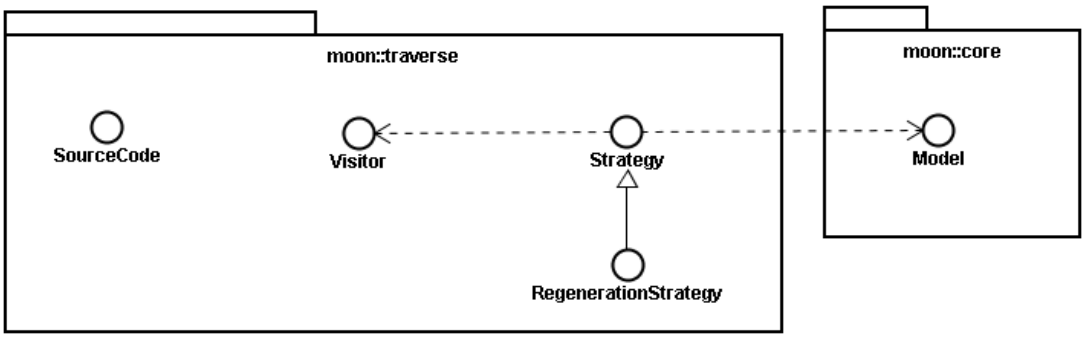

Figura 6.5: Diagrama de clases de regeneración del código

Puesto que la regeneración es particular de cada lenguaje, este módulo de regeneración del código al igual que el módulo de análisis del código, es dependiente de la instanciación concreta, y por *ende* del lenguaje.

Esto se ve reforzado, puesto que dicha regeneración depende en último lugar de la organización física de componentes (ficheros fuentes, binarios, recursos, etc.) Este factor es habitualmente diferente en cada lenguaje donde se aplican reglas específicas que impiden una estrategia de recuperación común, por ejemplo:

- la estructura lógica y física está estrechamente ligada en un lenguaje como Java, donde una clase pública tiene una correspondencia uno a uno con su fichero fuente. Debemos

señalar que en el fichero fuente puede incluir más clases pero con otros modificadores de acceso diferentes.

- en .NET esta correspondencia está abierta, pudiendo un mismo fichero físico contener varias clases o una clase estar definida en varios ficheros físicos.
- en EIFFEL, la utilización de un *cluster* a la hora de organizar las clases se hace con ficheros externos de recursos.
- etc.

Así pues, ya que cada lenguaje, entorno o plataforma puede dar diferentes directrices y normas particulares, es necesario que este módulo introduzca dependencias particulares del lenguaje.

6.5. Conclusiones al uso de *frameworks* como solución de soporte

La utilización de *frameworks* da una solución de soporte a la refactorización de código con un cierto grado de independencia del lenguaje. Aunque es necesario particularizar ciertos aspectos para cada lenguaje, también se permite la reutilización de elementos ya implementados y disponibles en el núcleo del *framework*, para la resolución de problemas comunes en una familia de lenguajes.

Como se apuntó en [Markiewicz and de Lucena, 2001], a lo largo de los trabajos realizados se ha ido evolucionando de una solución basada en bibliotecas, hacia un *framework* de caja blanca [Crespo et al., 2004]. Dicho *framework* basa su funcionamiento en mecanismos de herencia propios del lenguaje de implementación[4]. Sin embargo, a lo largo del tiempo, se ha ido evolucionando la solución hacia herramientas que asistan al usuario en la utilización del *framework*, derivando hacia un *framework* de caja gris o negra [Marticorena and Crespo, 2008].

Ambas soluciones son válidas y funcionales, permitiendo una reutilización del *framework* desde el punto de vista de caja blanca, mediante la extensión de clases e implementación concreta de métodos abstractos (*hook methods*) así como la utilización de scripts, asistentes, *wizards*, etc, para su utilización como *framework* de caja gris o negra.

El hecho de evolucionar el motor en este sentido, confirma la validez del proceso seguido, puesto que los resultados finales siguen la lógica marcada en el desarrollo de *frameworks* y en su evolución a lo largo del tiempo, según se adquiere una cierta experiencia y madurez en el desarrollo de los mismos.

En el siguiente capítulo, se documentará un caso de estudio de instanciación del *framework* sobre el lenguaje JAVA, para la implementación y ejecución de refactorizaciones, demostrando la aplicabilidad de la solución propuesta.

[4]El prototipo actual ha sido desarrollado en JAVA.

CAPÍTULO 7

CASO DE ESTUDIO

En este capítulo se muestran los resultados obtenidos en la implementación de las anteriores propuestas, sobre uno de los lenguajes más extendidos de la familia de LPOO, como es JAVA. En concreto, se toma JAVA como lenguaje de referencia dada su posición dominante en el mercado, dentro de la familia de LPOO. Por otro lado, tiene una serie de propiedades muy diferentes al lenguaje EIFFEL, que ya fue objeto de estudio previo en [Crespo, 2000]. Tanto desde el punto de vista de herencia, y muy en particular desde el punto de vista de la genericidad, es un lenguaje ideal para estudiar su posible soporte con MOON.

La extensión concreta al metamodelo se denomina JAVAMOON y se detalla su estudio y construcción en la Sec. 7.1. Una vez que se tiene una metamodelo como extensión concreta de MOON para el lenguaje objetivo de la refactorización, dando soporte a la extracción de código que se estableció en el Capítulo 6, se implementan las distintas funciones, predicados y acciones que completan los repositorios correspondientes, permitiendo la extracción y modificación de la información del código.

Para validar la construcción de las refactorizaciones, se construye un prototipo en una serie de iteraciones [Fuente de la Fuente and Herrero Paredes, 2008, Fuente de la Fuente, 2009, Gómez San Martín and Mediavilla Saiz, 2011] que incluye un asistente que guía el proceso, tal y como se describe en la Sec. 7.2. Con la refactorización ya construida e integrada en el prototipo utilizando la solución de *framework* de caja negra, se aplica por parte del usuario, tal y como se describe en la misma sección.

El conjunto de refactorizaciones implementadas, se describe en la Sec. 7.3, indicando sus características y particularidades encontradas en cada caso, a lo largo de los trabajos realizados. En la Sec. 7.4, se detalla el proceso seguido para regenerar el código JAVA refactorizado. Finalmente, en la Sec. 7.5, se concluye con un análisis DAFO[1] del prototipo.

[1] Análisis o matriz de **D**ebilidades, **A**menazas, **F**ortalezas y **O**portunidades.

7.1. Extensión de JAVA para el metamodelo MOON: JAVAMOON

A partir del metamodelo MOON, se analiza la especialización del metamodelo asociado para un lenguaje concreto. Se ha elegido JAVA por ser un LPOO, ampliamente extendido y que por otro lado incluye características adicionales no consideradas en MOON. Esto permite validar la adecuación de la solución basada en *frameworks*, que permite trabajar a un nivel abstracto (metamodelo MOON) o bien a un nivel concreto (metamodelo JAVAMOON) en la construcción y aplicación de refactorizaciones sobre código concreto y real.

A continuación se describen las distintas clases incluidas en la extensión de MOON para dar soporte al lenguaje JAVA. El orden establecido en las secciones es el mismo que el seguido en el Capítulo 3, en su Sec. 3.1, con la salvedad de la inclusión de las anotaciones (ver Apartado 7.1.8).

7.1.1. Tipos y Clases

En JAVAMOON se añaden los conceptos de `JavaPackage`, `JavaClassDef` y `JavaType` (ver Fig. 7.1) como extensiones naturales de MOON. Las relaciones entre dichas clases se establecen a través de sus ancestros correspondientes en MOON, con la salvedad de la inclusión de una posible pertenencia de una clase a otra existente en JAVA, modelando las clases anidadas e internas, conceptos no contemplados en MOON.

Sin embargo es necesario introducir elementos adicionales, debido a la sintaxis y semántica particular de JAVA:

- `JavaEnum` y `JavaAnnotation` como conceptos adicionales incluidos en JAVA con una sintaxis y semántica particular para modelar tipos enumerados y anotaciones respectivamente [McLaughlin and Flanagan, 2004, Gosling et al., 2005].
- `JavaImport` para establecer las relaciones de importación entre clases y tipos en el código.
- Variantes de tipos: *arrays* (`JavaArrayType`), *arrays* genéricos (`JavaGeneric-ArrayType`), tipos desconocidos (? en genericidad) (`JavaWilcardType`) y un decorador (`JavaDecoratorType`) (siguiendo el patrón de diseño **DECORATOR** [Gamma et al., 1995]) que permita almacenar el estado cualificado o no cuando se referencia a un tipo.

La inclusión de la enumeraciones en JAVAMOON, se simplifica al cumplir la semántica de una clase, con la salvedad de que algunos de sus atributos son constantes del tipo enumerado `JavaEnumConstant` (ver Fig. 7.3). La semántica de la definición y uso de la anotaciones es bastante más compleja dándose una descripción más detallada en el Apartado 7.1.8.

Por otro lado, dada la naturaleza particular de JAVA, al tratarse de un lenguaje híbrido no puramente orientado a objetos, hace que el conjunto de tipos primitivos utilizados por valor,

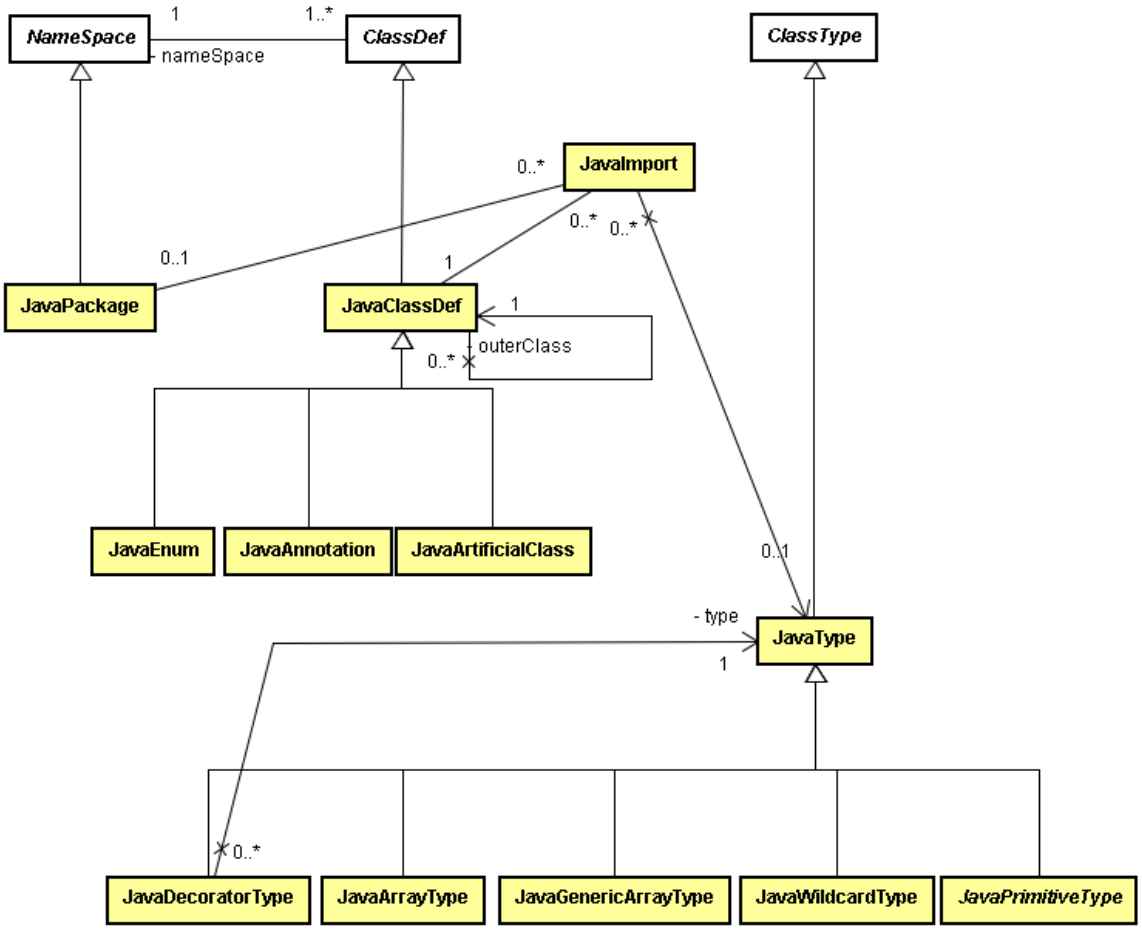

Figura 7.1: Tipos y clases en JAVAMOON

y no por referencia, cobren un papel muy importante. Este conjunto de tipos se ha modelado utilizando clases descendientes de la superclase `JavaPrimitiveType` (ver Fig. 7.2).

7.1.2. Entidades

En primer lugar se tiene un conjunto básico de entidades como extensión de las entidades ya clasificadas en MOON (ver Fig. 7.3), con la inclusión adicional del concepto de las constantes en tipos enumerados, como extensión del concepto de atributo.

Sin embargo, aparecen un número elevado de entidades no clasificables como subtipos de las ya definidas en MOON, necesarias para incluir al completo el código escrito en JAVA (ver Fig. 7.4). Estas clases se definen como descendientes directos de `Entity`:

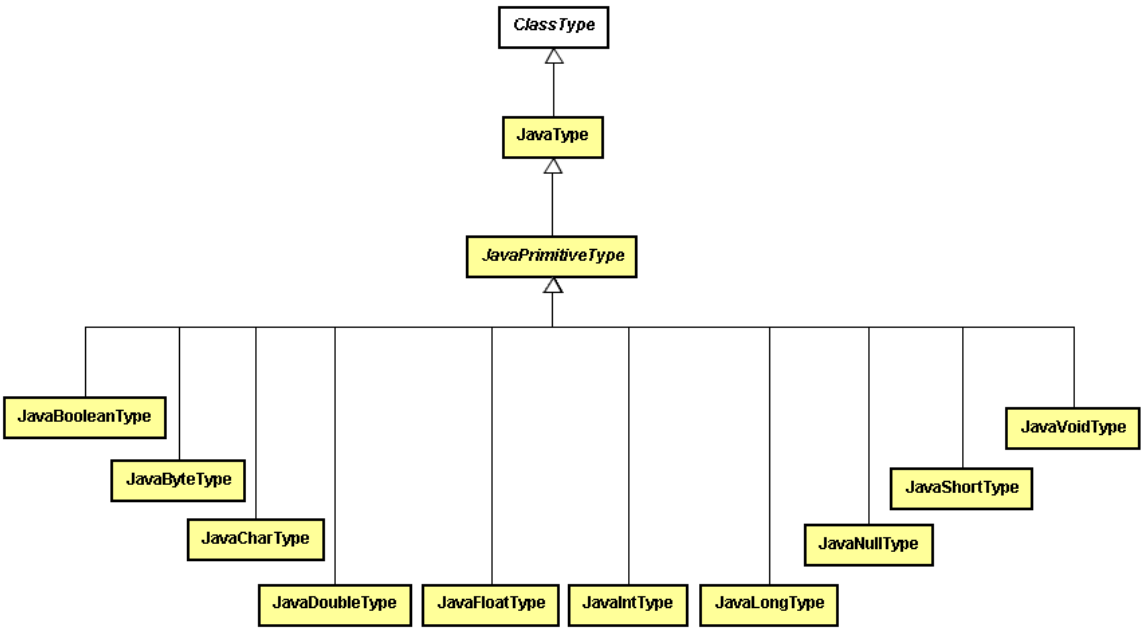

Figura 7.2: Tipos primitivos en JavaMOON

- `JavaStaticEntity`: acceso de propiedades estáticas a través del propio nombre de la clase.
  Ejemplo: `System.gc();` o `PrintStream o = System.out;`
- `JavaOperator`: operador utilizado en operaciones aritméticas en infijo.
- `JavaArtificialEntity`: permite construir entidades artificiales, que no existen en el código, pero que son necesarias construir para ligar a una expresión. Como especializaciones de dicha clase se tienen:
  - `JavaArtificialEntityDefaultConstructor`: los constructores por defecto no existen en el código, sino que se crean por la ausencia de otros constructores.
    Ejemplo: `new Constructor();`
  - `JavaArtificialLengthEntity`: en Java los arrays tienen un atributo `length`. Dado que no existe una clase asociada al tipo *array* se crea como una entidad artificial.
    Ejemplo: `while (a < array.length){ a++; }`
  - `JavaArtificialEntityManifestConstant`: se permite el acceso de entidades y métodos a partir de entidades que contienen constantes manifiestas.
    Ejemplo: `"text".toArray();`

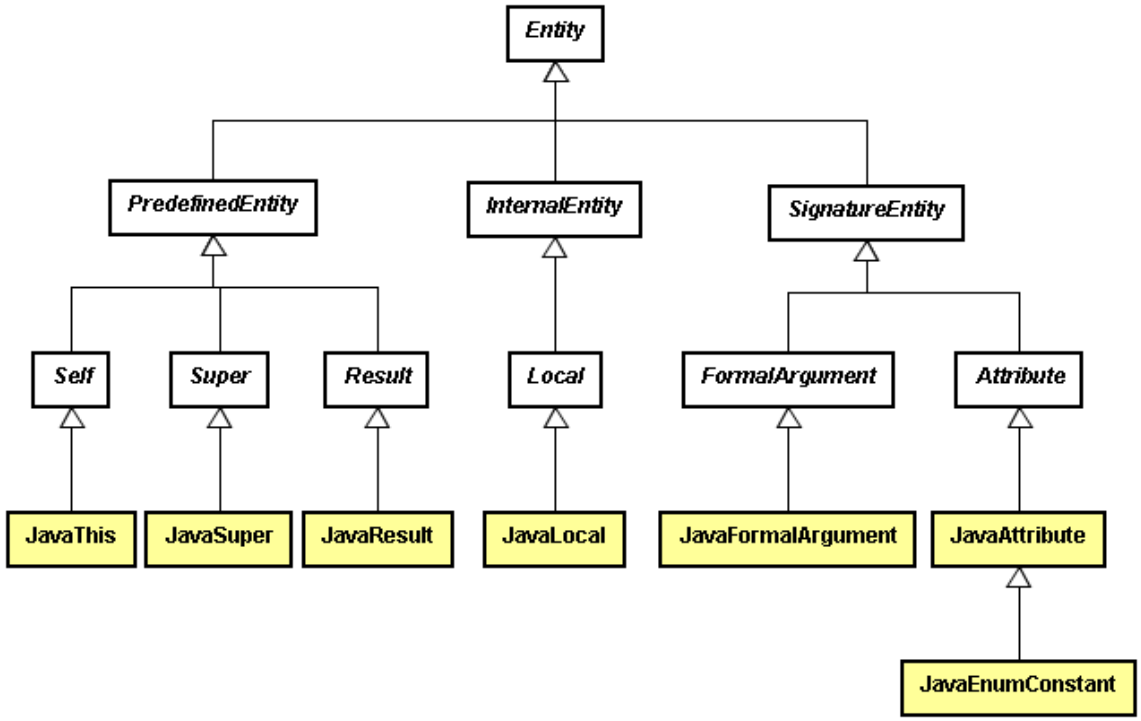

Figura 7.3: Entidades en JAVAMOON

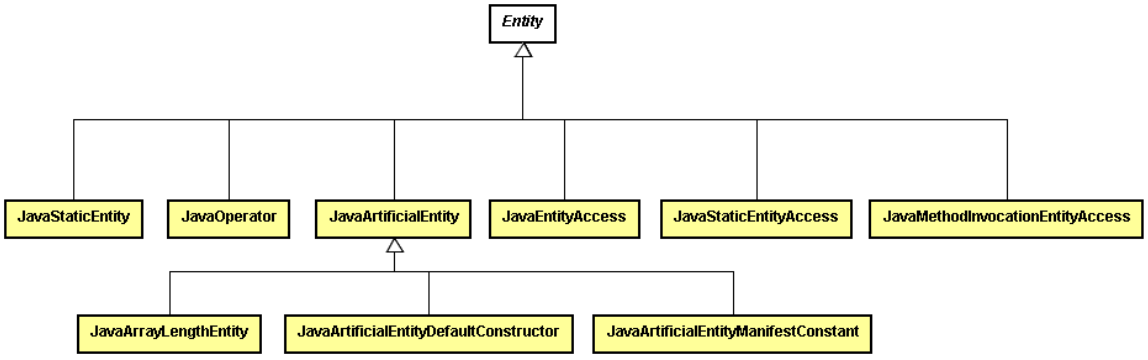

Figura 7.4: Entidades artificiales en JAVAMOON

- `JavaEntityAccess`: acceso a valores en un array a partir de entidades no estáticas. Ejemplo: `this.c[0]; //c array in this`

- `JavaStaticEntityAccess`: acceso a valores en un array a partir de entidades estáticas. Ejemplo: `A.c[0]; //c static array in A`

- `JavaMethodInvocationEntityAccess`: acceso a valores en un array a partir de invocaciones a métodos. Ejemplo: `this.m()[0]; //m()returns an array`

Como se puede comprobar, el modelado del acceso a *arrays* y sus expresiones asociadas, añade una complejidad adicional, puesto que su sintaxis y semántica va más allá de lo contemplado inicialmente en MOON dada su naturaleza de lenguaje minimal.

7.1.3. Métodos

El soporte de métodos en JAVA no varía respecto a lo planteado en MOON (ver Fig. 7.5), con clases concretas que implementan los conceptos de rutina y función.

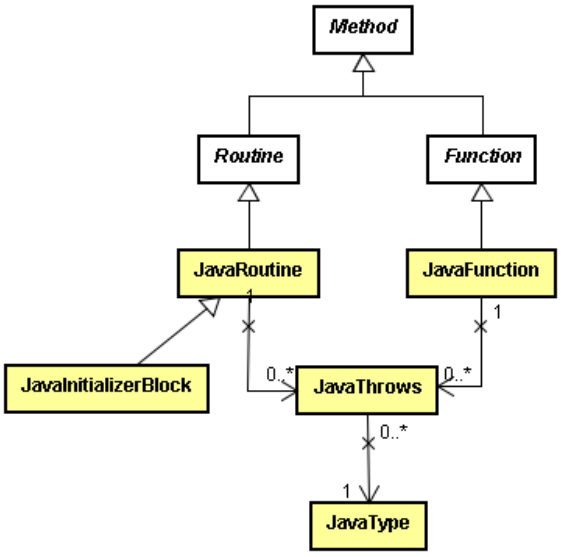

Figura 7.5: Métodos en JAVAMOON

Dado que en MOON no se contempla el tratamiento de excepciones, se añade el concepto de cláusula **`throws`** que permite ligar un método al conjunto de tipos (excepciones en JAVA) que pueden ser lanzadas desde el cuerpo del mismo.

También se añade el concepto de bloque de inicialización (`JavaInitializerBlock`), existente en JAVA, que no se referencia a través de ningún nombre, y que cumplen el rol de rutinas de inicialización de atributos (extienden de `JavaRoutine`). Los bloques de inicialización pueden ser estáticos o no estáticos.

7.1.4. Genericidad

El modelado de la genericidad se limita a extender la variante de subtipado o conformidad, puesto que en JAVA no existe acotación por cláusulas tal que (ver Fig. 7.6). Se incluyen las dos variantes de acotación en sentido covariante (con **`extends`**) y en sentido contravariante (con **`super`**) [Gosling et al., 2005, Naftalin and Wadler, 2007].

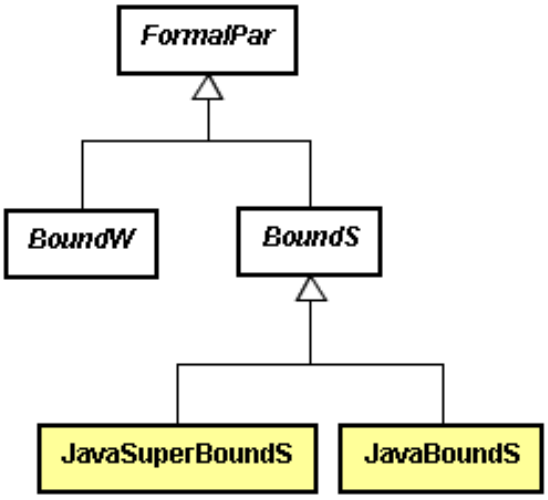

Figura 7.6: Genericidad en JAVAMOON

- `JavaBoundS`: acotación en el sentido covariante.
  Ejemplo: `<E extends Comparable<E>>`
- `JavaSuperBoundS`: acotación en el sentido contravariante.
  Ejemplo: `<E super ArrayList<E>>`

7.1.5. Herencia

El modelado de la herencia se simplifica (ver Fig. 7.7), al estar definido de manera casi completa en MOON, con la salvedad de la semántica específica en JAVA respecto a la restricción de herencia simple de clases y la implementación múltiple de interfaces.

- `JavaExtends`: cláusula de herencia entre clases y entre interfaces.
  Ejemplo1: `class A extends B {`
  Ejemplo2: `interface A extends B {`
- `JavaImplements`: cláusula de herencia de una clase a una interfaz.
  Ejemplo: `class A implements B,C {`

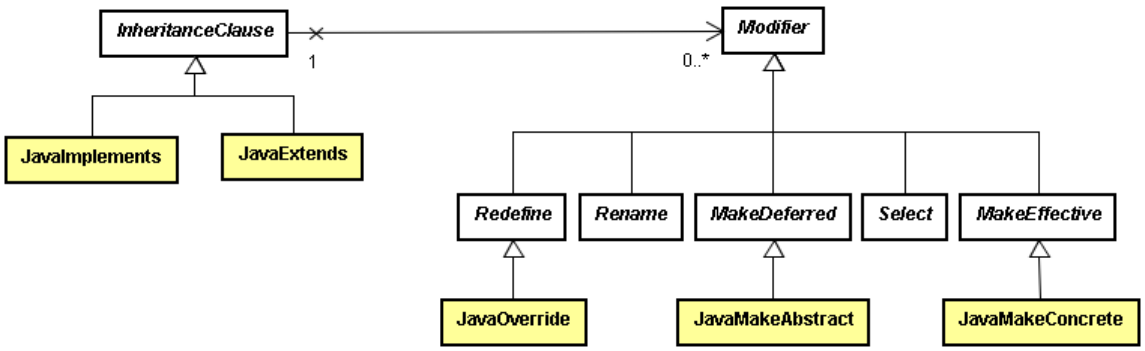

Figura 7.7: Herencia en JavaMOON

7.1.6. Expresiones e instrucciones

La parte más compleja de modelar es la extensión de instrucciones y expresiones, puesto que las simplificaciones realizadas en MOON, obligan a que gran parte de la sintaxis y semántica particular sea almacenada en la extensión concreta del lenguaje. Pero por otro lado, se intenta seguir manteniendo un número reducido de clases para definir un metamodelo lo más simple posible.

MOON no está orientado al almacenamiento detallado de la información del código, y en particular del cuerpo de los métodos. Sin embargo, sí que se almacena la información básica sobre entidades utilizadas, construcciones, invocaciones y asignaciones realizadas sobre las entidades. Esto permite un análisis completo en la mayoría de los casos, pero a la hora de representar el código es necesario introducir elementos adicionales.

En el modelado de expresiones, aparece un nuevo conjunto de expresiones (ver Fig. 7.8):

- `JavaCallExprAssignment`: asignación del valor de una expresión a otra.
  Ejemplo: `a = b = c;`
- `JavaCallExprCast`: moldeado explícito de tipos.
  Ejemplo: `A a = (A)x;`
- `JavaCallExprConditionalExpression`: operador condicional `?`.
  Ejemplo: `r = a == 0 ? x : y;`
- `JavaCallExprArrayAccess`: acceso a valores de un *array* a través de sus índices.
  Ejemplo: `x = a[0][1];`
- `JavaCallExprArrayCreation`: instanciación para la reserva de memoria de un *array*.
  Ejemplo: `new int[10];`

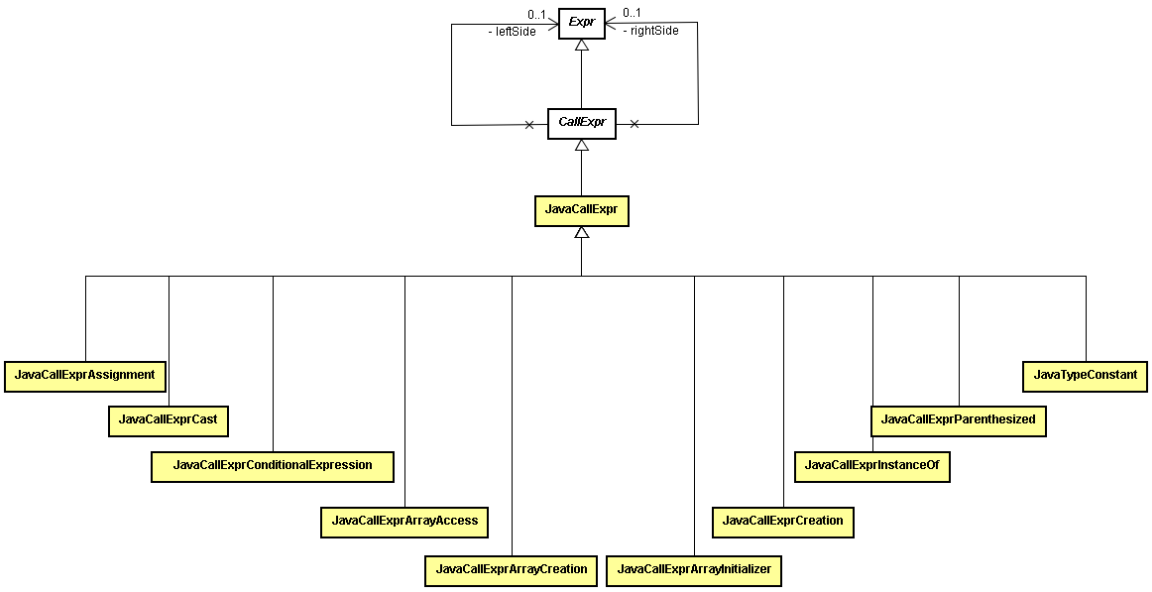

Figura 7.8: Expresiones en JAVAMOON

- `JavaCallExprArrayInitializer`: inicialización de *arrays* con literales.
  Ejemplo: `int[] a = {1,2,3};`
- `JavaCallExprCreation`: expresión de creación de instancias. En MOON la creación está ligada a una instrucción, pero en el caso de JAVA está ligada a una expresión usada como parte derecha de una asignación.
  Ejemplo: `String s = new String("text");`
- `JavaCallExprInstanceOf`: expresión *booleana* resultado de utilizar el operador **`instanceof`**.
  Ejemplo: `boolean b = a instanceof B;`
- `JavaCallExprParenthesized`: decorador para permitir el anidamiento de paréntesis.
  Ejemplo: `a =(b + c)/(10 + (8*9));`
- `JavaTypeConstant`: uso de un literal de clase. Dada la naturaleza particular de este tipo de expresiones en JAVA, ligadas a una entidad de tipo `JavaStaticEntity` (ver Fig. 7.4), no son consideradas como constantes manifiestas puesto que una constante no está vinculada a una entidad de código.
  Ejemplo: `String.class`

En MOON se planteó la eliminación de expresiones binarias y unarias asumiendo que se pueden reescribir como una expresión de envío mensajes. Por ejemplo, `a + b` deberá

reescribirse como `a.+(b)`. Aunque esta solución es válida para el análisis de las expresiones, puesto que se manejan a un nivel abstracto como `CallExpr` en el metamodelo MOON, para la recuperación completa del texto, es necesario almacenar información adicional sobre el operador y la notación utilizada (prefijo, infijo o postfijo). Como se puede ver en la Fig. 7.9, se realiza una especialización de las expresiones (`JavaCallExpr`), almacenando el operador y el tipo de notación, para realizar una posterior recuperación correcta.

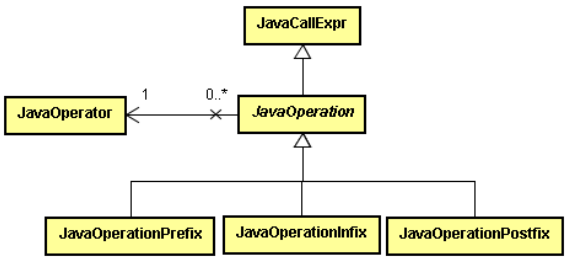

Figura 7.9: Operaciones en JavaMOON

El uso de constantes se extiende en Java, incluyendo extensiones para la totalidad de constantes ya definidas en MOON (ver Fig. 7.10), ampliando la semántica cuando es necesario, así como incluyendo tres tipos especiales de constantes:

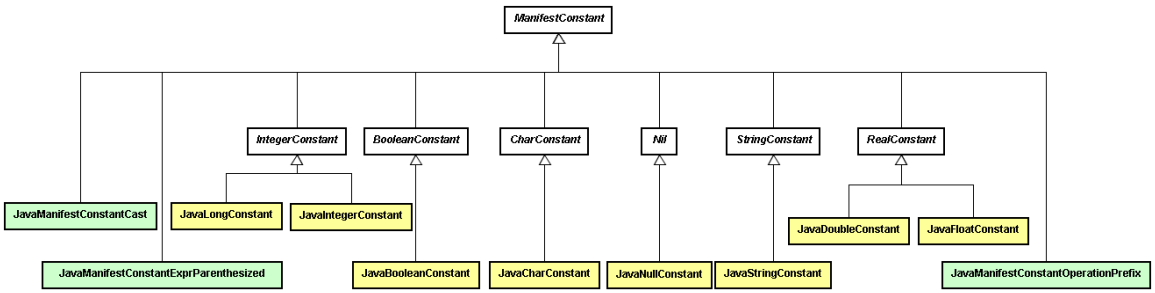

Figura 7.10: Constantes en JavaMOON

- `JavaManifestConstantExprParenthesized`: constante manifiesta contenida entre paréntesis.
  Ejemplo: `("text")`

- `JavaManifestConstantOperationPrefix`: operador prefijo delante de expresiones contantes.
  Ejemplo: `-5;`

- `JavaManifestConstantCast`: permite el uso de moldeado con constantes manifiestas.
  Ejemplo: `(long)1000;`;

En MOON se definió una jerarquía paralela de clases descendientes de expresiones entre `Expr`, `CallExpr` y `ManifestConstant` (ver Capítulo 3, Fig. 3.6). En ausencia de herencia múltiple de clases (característica en los lenguajes de la corriente principal), se ha seguido manteniendo una jerarquía paralela en las expresiones. En particular en cuanto al uso de paréntesis o moldeados en función de que se apliquen a llamadas a expresión o uso de constantes manifiestas (ver Fig. 7.11).

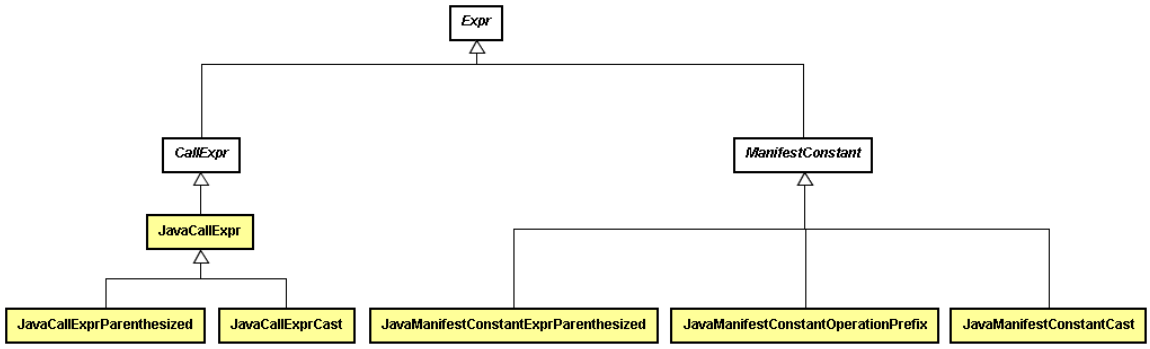

Figura 7.11: Jerarquía paralela de llamadas a expresiones y uso de constantes manifiestas en JAVAMOON

En el caso de las instrucciones, la jerarquía básica de MOON se reutiliza, extendiendo las clases ya definidas (ver Fig. 7.12), con la excepción de `CreationInstr`. En JAVA no existe una instrucción de creación como tal, sino que la instanciación de objetos siempre está ligada a una expresión obtenida con el operador **new**. Esto se ha modelado previamente mediante la expresión `JavaCallExprCreation`.

Sin embargo, junto con la extensión natural de las clases del metamodelo MOON, es necesario añadir un conjunto adicional de clases para almacenar aquella información necesaria no almacenada en MOON:

- `JavaInstrNoMoon`: en la búsqueda de un lenguaje minimal, muchas de las construcciones contenidas en el lenguaje a analizar no tienen su correspondiente reflejo en el metamodelo MOON. Estas instrucciones carecen de interés al no contener información sobre tipos, entidades o expresiones, por lo que simplemente se debe almacenar su texto asociado. Por otro lado, es fundamental no perder ls información para poder regenerar el código refactorizado. Por lo tanto, esta clase almacena la información relativa a las construcciones sintácticas habituales en muchos lenguajes así como las particulares al mismo.

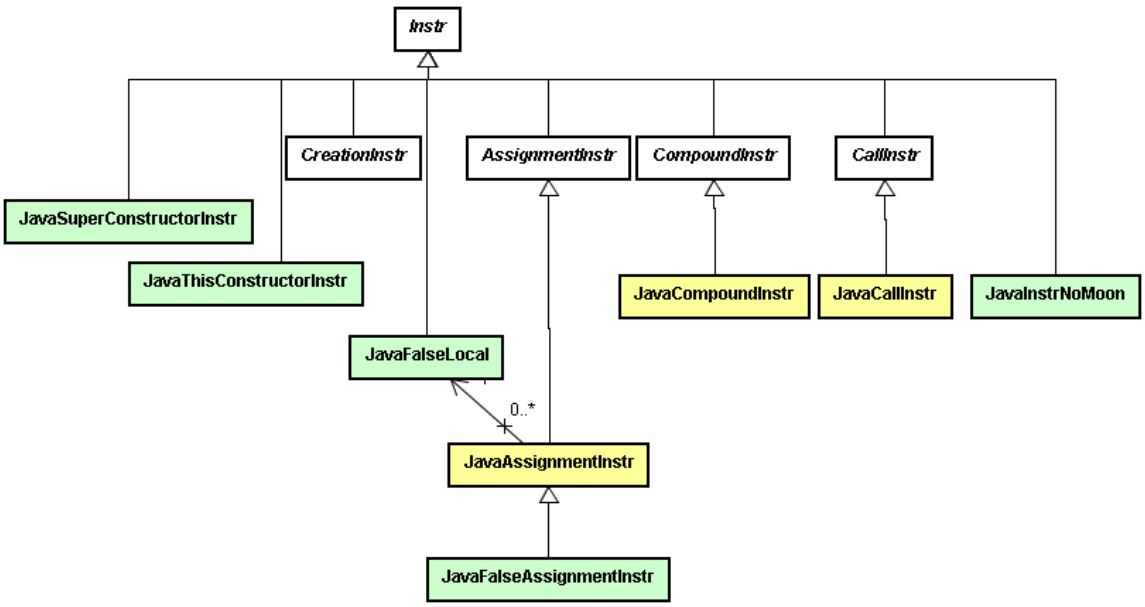

Figura 7.12: Instrucciones en JavaMOON

Ejemplo: **assert, if, for, switch, while**, etc.

- `JavaSuperConstructorInstr`: invocación al constructor de las superclase.
  Ejemplo: **super**`(a,b);`
- `JavaThisConstructorInstr`: invocación a otro constructor de la clase.
  Ejemplo: **this**`(a,b);`
- `JavaFalseLocal`: referencia a una declaración de variable local en el código. Su finalidad es permitir la recuperación del código de la variable.
- `JavaFalseAssignmentInstr`: dado que el cuerpo de los métodos se deben almacenar como conjunto de instrucciones, esta clase permite crear instrucciones de asignación, que almacenan la información necesaria. En particular, dado que MOON no almacena información de las sentencias condicionales, estas asignaciones "ficticias" son utilizadas para guardar información de las expresiones *booleanas* utilizadas en Java en casi todas sus sentencias condicionales y asertos, o bien asignando su valor a entidades artificiales (`JavaArtificialEntity`) (ver Fig. 7.4).

  Ejemplos:

```
if (a > 10)
for(;a < 10;)
assert x == null;
```

7.1.7. Comentarios de código

En el metamodelo MOON se incluyó este concepto a un nivel abstracto. JAVA, como la totalidad de LPOO, incluye y extiende el concepto de comentario de código, con algunas variaciones. En este caso concreto, se ha resuelto clasificando los comentarios en tres tipos (ver Fig. 7.13):

- `JavaCommentSingleLine`: comentario de línea de una línea utilizando `//` como marcador de principio de comentario.
- `JavaCommentMultiLine`: comentario de varias líneas indicando el principio con `/*` y el final con `*/`.
- `JavaCommentJavadoc`: comentario para la generación de documentación con la herramienta `javadoc` u otras herramientas similares. Los comentarios javadoc empiezan con `/**` y finalizan con `*/`. Pueden contener etiquetas especiales *e.g.* `@author`, `@version`, `@param`, `@return`, etc.

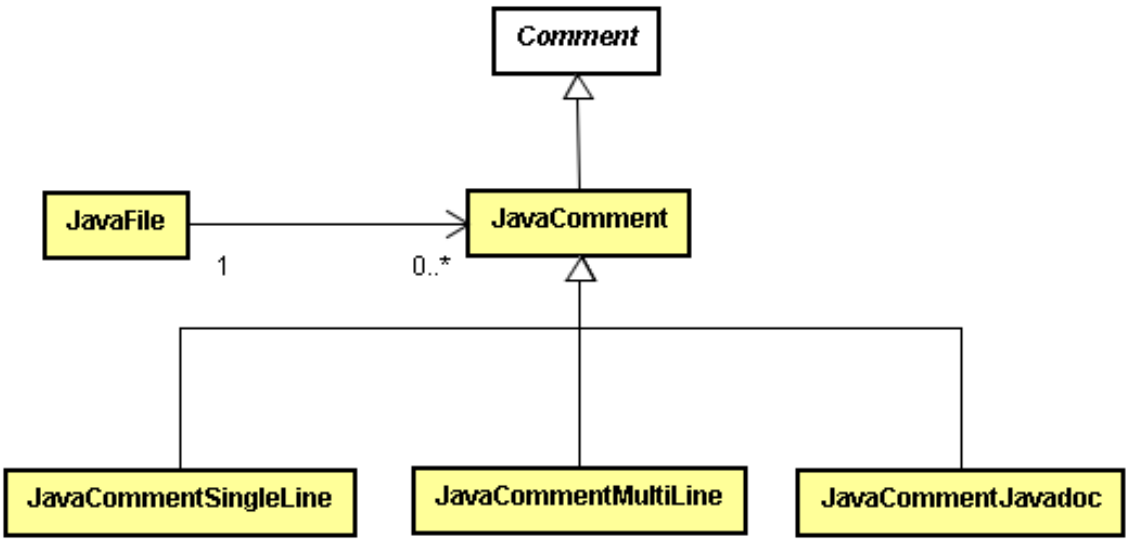

Figura 7.13: Comentarios en JAVA

Todas estas clases heredan de la clase del `Comment` e indirectamente de la clase raíz `ObjectMoon`, para mantener sus posiciones dentro del código almacenado. No se incluye ninguna cuestión semántica sobre la relación de los comentarios con los distintos elementos del metamodelo JAVAMOON. Los comentarios están contenidos en los ficheros fuente modelados en la clase `JavaFile`.

7.1.8. Anotaciones

Las anotaciones son un elemento incluido en las últimas versiones del lenguaje [McLaughlin and Flanagan, 2004, Gosling et al., 2005] y que es previsible que su uso se amplíe en futuras versiones [Oracle, 2011].

La facilidad de poder programar sus propias anotaciones, y de poder controlar el procesado de las mismas a través de herramientas del *kit* de desarrollo abren un gran abanico de posibilidades. Como ejemplo se puede citar el gran cambio sufrido en las últimas versiones de Java Enterprise Edition (JEE), al cobrar una importancia vital el uso de anotaciones.

La inclusión de dichos elementos en el metamodelo JavaMOON ha sido de una manera suave (ver Fig. 7.14). En cuanto a su construcción, las anotaciones (`JavaAnnotation`) son consideradas una especialización de las clases Java (`JavaClassDef`) heredando sus propiedades, con la salvedad que mantienen un mapa de asociaciones entre sus métodos (`Method`) y su valores por defecto (`Expr`).

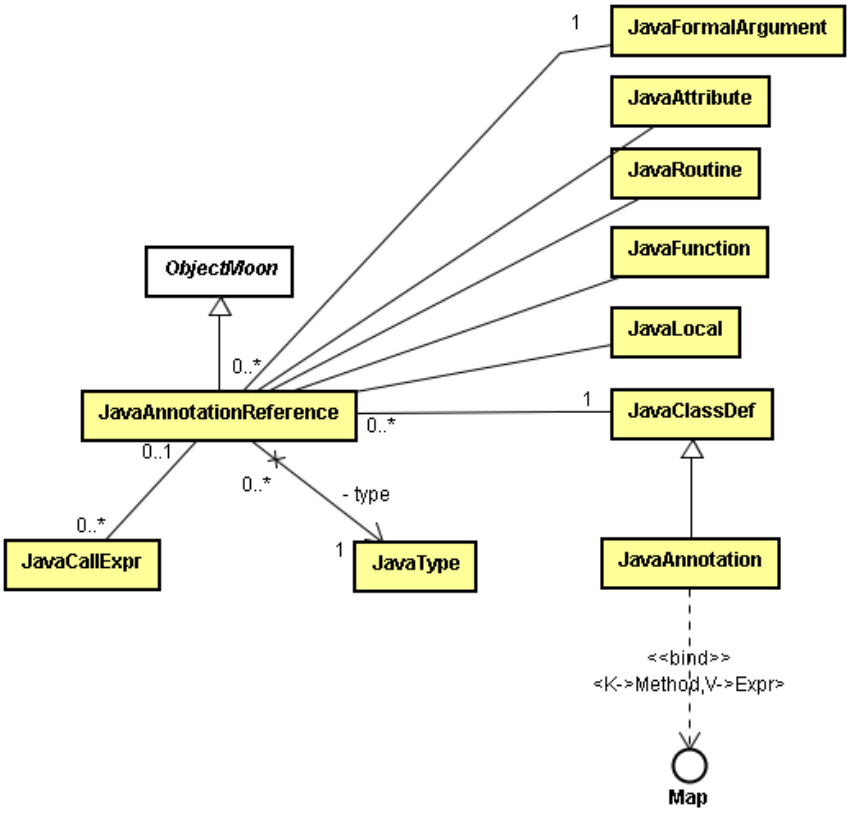

Figura 7.14: Anotaciones en JavaMOON

En cuanto a su uso en el código, se ha introducido un nuevo elemento para referenciar el uso de la anotación (`JavaAnnotationReference`) que vincula la anotación (el tipo de la misma, generado por su clase determinante). Dado que al ser utilizada una anotación, se puede requerir dar valores a sus propiedades se utilizan las expresiones (`JavaCallExpr`) para crear los correspondientes valores.

En este caso concreto sí que se ha vinculado el uso de las anotaciones a cada uno de los elementos particulares de JavaMOON que pueden contener anotaciones, para tener toda la información semántica de su uso. Es previsible que en base a futuras modificaciones del lenguaje [Oracle, 2011], se asocien a más elementos del metamodelo.

7.1.9. Conclusiones al metamodelo JavaMOON

Con todo el conjunto de clases descrito en estas secciones previas, se puede representar el código Java, para su posterior refactorización y regeneración del código. El núcleo del metamodelo de Java para MOON (JavaMOON) consta finalmente de un total de unas 110 clases e interfaces. El núcleo es una extensión de MOON, sin incluir las clases encargadas de extraer ni de regenerar el código.

Aunque el número pueda ser elevado, frente a las 50 clases e interfaces del metamodelo MOON, hay que considerar la simplicidad del código de estas clases, puesto que en la mayoría de casos gran parte de la funcionalidad se hereda del metamodelo de MOON.

La generación de los objetos del metamodelo JavaMOON a partir del código existente, se realiza en dos fases:

- **Carga binaria**: se extrae la información de los objetos contenidos físicamente en ficheros `.jar` o ficheros `.class`. La información se recupera a través de mecanismos de reflexión incluidos en el lenguaje Java. Toda esta información es de sólo lectura, puesto que el código binario no será transformado.

- **Carga de ficheros fuentes**: se reutiliza la implementación existente en Eclipse del patrón de diseño **Visitor** aplicado a AST, realizando tres implementaciones diferentes del mismo. Mediante tres recorridos se extrae: en primer lugar la información de tipos, en segundo lugar el esqueleto de las clases y por último se completa el cuerpo de los métodos.

7.2. Desarrollo actual

Como prueba de concepto, se ha desarrollado finalmente un prototipo [Gómez San Martín and Mediavilla Saiz, 2011] con un conjunto reducido de refactorizaciones. El conjunto fue previamente implementado usando bibliotecas, donde cada refactorización es codificada e implementada sobre dicha base. Los cambios a las refactorizaciones eran realizados en un nuevo proceso de desarrollo, editando el código fuente y recompilando las nuevas refactorizaciones.

Como ya se indicó en la evolución natural de los *frameworks*, en este caso también se evolucionó desde una solución imperativa basada en mecanismos de herencia hacia una solución declarativa utilizando ficheros XML. La solución actual fue presentada en [Marticorena et al., 2011a].

La herramienta está desarrollada en JAVA como plugin del entorno de desarrollo integrado ECLIPSE, pero la solución mostrada podría ser aplicada con otros lenguajes que incluyan soporte a XML, y capacidades de programación con reflexión. Asimismo la solución puede ser extrapolable a diferentes plataformas.

Con la solución basada en XML ya no es necesario codificar las refactorizaciones manualmente, sino que la refactorización se define, y los elementos constituyentes son ensamblados en tiempo de ejecución a partir de las implementaciones ya existentes en los repositorios.

Sin embargo, siempre es necesaria la existencia de un conjunto completo o suficiente de consultas y transformaciones, pero si este requisito se cumple, la refactorización es construida en un tiempo muy breve. En nuestro caso, la tarea se simplifica, en tanto en cuanto que los repositorios han sido completados en varias iteraciones.

7.2.1. Construcción de refactorizaciones

Para evitar errores manuales en la edición de las refactorizaciones, o en lo que sería su fichero XML correspondiente, se provee de un asistente o *wizard* para su construcción. La herramienta guía todo el proceso, mostrando los elementos actuales en los metamodelos y en los repositorios, permitiendo su selección y enganche.

Aunque no hay diferencia final de resultados entre ambas soluciones, ya sean refactorizaciones codificadas a mano o generadas con el asistente, el problema de su evolución y mantenimiento es resuelto en un modo más simple. Los posibles futuros cambios son aplicados usando nuevamente el asistente, en pocos segundos. Mientras que otras soluciones implican la interacción del programador con alto conocimiento del IDE, API y el lenguaje de implementación utilizado.

Paso 1: Descripción de la refactorización

En este primer paso se debe rellenar la información general de la refactorización, que corresponde con: nombre, descripción, imagen, motivación, palabras clave y categorías (ver Fig. 7.15). La estructura es similar a la planteada en el Capítulo 4, aunque ampliando alguna cuestión gráfica como la imagen y palabras claves o categorías para facilitar su indexación y consulta en el *plugin*.

Paso 2: Entradas

En el segundo paso se configurarán las entradas de la refactorización, para cada entrada se definirá los campos: *name*, *main*, *from*, *method* (ver Fig. 7.16). La lista de la derecha,

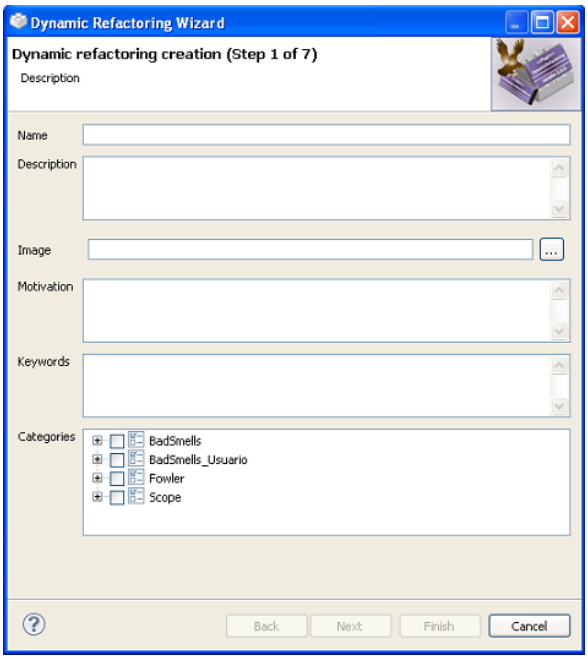

Figura 7.15: Paso 1: Descripción de la refactorización

`Inputs`, muestra las entradas que ya han sido seleccionadas para la refactorización y la que se encuentra a la izquierda, `Types`, la lista de tipos disponibles.

Tal y como se planteó en la caracterización de las refactorizaciones (ver Sec. 4.2), la entrada principal junto con las entradas adicionales, tienen un efecto directo en el diseño de la interfaz. Por un lado, el *plugin* sólo mostrará posteriormente como aplicables aquellas refactorizaciones cuyo tipo de entrada principal (*main*) coincide con el elemento seleccionado por el usuario en ese momento. En segundo lugar, la interfaz gráfica generada para introducir el resto de entradas adicionales se determina, y crea de forma dinámica a partir de dichas entradas.

El metamodelo permite definir ciertas entradas en función de los valores de otras, a través de los mecanismos de reflexión, utilizando los valores de origen (*from*) y el método de consulta a aplicar sobre dicho objeto (*method*).

Paso 3: Precondiciones

En el tercer paso se deberán seleccionar las precondiciones elegidas para que formen parte de la definición de la refactorización (ver Fig. 7.17). La lista de la derecha muestra las precondiciones que ya han sido seleccionadas para la refactorización y la que se encuentra a la izquierda, `Preconditions`, la lista de todas las precondiciones disponibles.

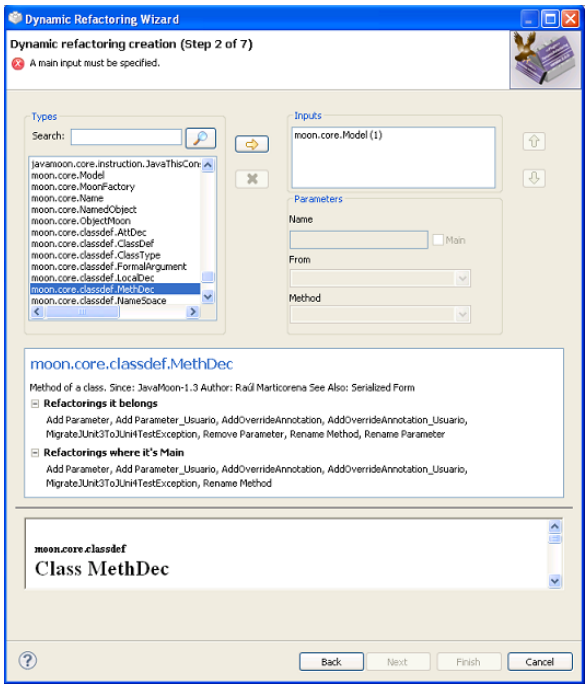

Figura 7.16: Paso 2: Entradas

En este caso se acude a los repositorios definidos para MOON, si son independientes del lenguaje, o para el lenguaje objetivo, en este caso JavaMOON.

Los argumentos de las precondiciones, así como en los siguientes pasos con las acciones y postcondiciones, se deben poder deducir o calcular a partir de las entradas de la refactorización.

Paso 4: Acciones

La interfaz correspondiente a este cuarto paso es idéntica a la del paso anterior diferenciándose únicamente en que en este paso se está trabajando con acciones en vez de precondiciones (ver Fig. 7.18). Por tanto, la forma de utilización es similar a la del paso anterior, siendo requisito indispensable que la acción a aplicar esté en alguno de los repositorios.

Paso 5: Postcondiciones

De la misma manera que en el paso anterior, la interfaz correspondiente es similar a la introducción de precondiciones y acciones, pero en este caso se seleccionan las postcondiciones. Éstas se verificarán posteriormente a la ejecución de la refactorización por parte del *framework*.

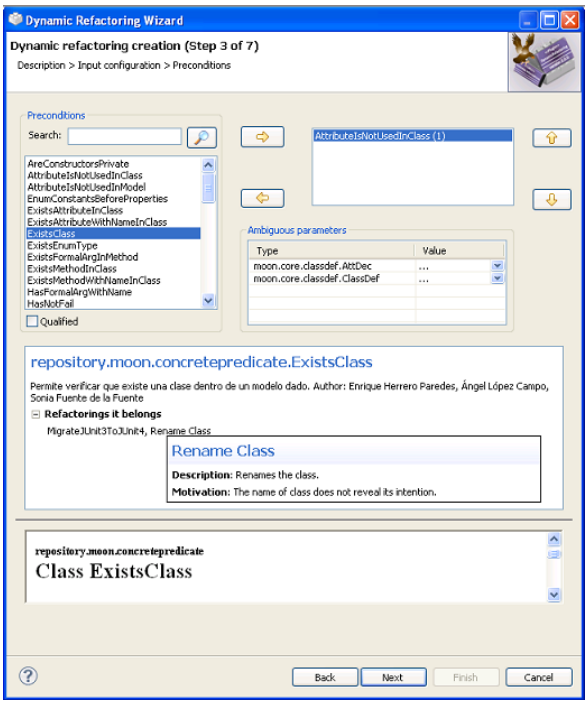

Figura 7.17: Paso 3: Precondiciones

Paso 6: Ejemplos

La sexta página del asistente permite asociar dos ejemplos de código a la definición de la refactorización, mostrando el estado previo y posterior a la refactorización. No es obligatorio introducir esta información, pero es recomendable, puesto que ha herramienta visualiza estos ejemplos en posteriores consultas por parte del usuario.

Paso 7: Resumen

En el último paso se muestra un resumen informativo de la definición que se ha ido construyendo, utilizando el asistente, con todos los datos indicados por el usuario (ver Fig. 7.19). Una vez que se haya revisado la configuración de la refactorización, se puede guardar pulsando el botón `Finish`.

Como resultado de la confirmación del usuario, se genera la refactorización a través un fichero XML con la definición de la refactorización. Esto aporta varias ventajas:

- La definición de la refactorización reside en un único punto y puede ser modificada de nuevo por el asistente sin afectar en nada al resto de refactorizaciones, metamodelos o repositorios.

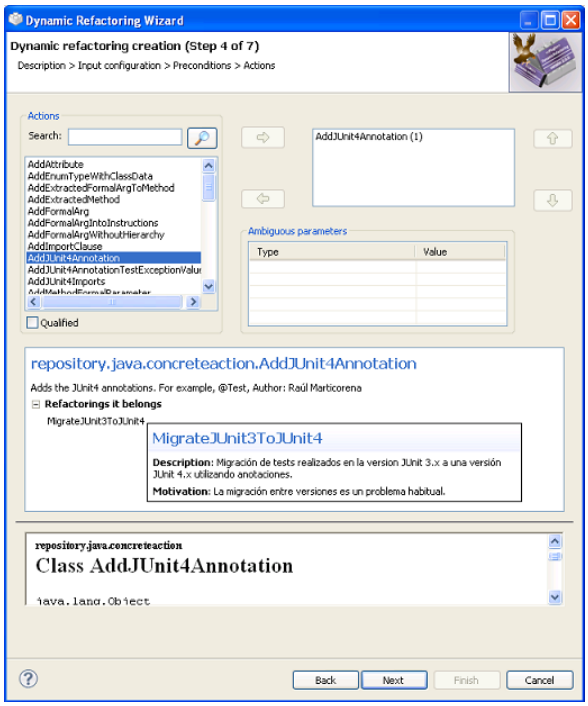

Figura 7.18: Paso 4: Acciones

- El carácter dinámico de la refactorización y su ensamblaje en ejecución, evita la necesidad de recompilación ante su modificación.
- La importación y exportación de refactorizaciones entre herramientas se restringe a detectar las dependencias del fichero XML con los distintos elementos de los repositorios, siempre y cuando las versiones de los metamodelos sean compatibles.

Definición XML construida

En el siguiente listado se muestra un ejemplo de una refactorización. En este caso, la refactorización **Rename Class** tal y como queda definida en el *plugin* actual, de forma declarativa en un fichero XML utilizando el asistente incluido en el prototipo:

Código 7.2.1: Fichero con la definición de la refactorización `Rename Class`

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE refactoring  SYSTEM 'refactoringDTD.dtd'>
<refactoring name="Rename Class">
        <information>
                <description>Renames the class.</description>
                <image src="renameclass.JPG"/>
                <motivation>The name of class does not reveal its intention.</motivation>
        <categorization>
            <classification name="Scope">
                <category>Class</category>
            </classification>
```

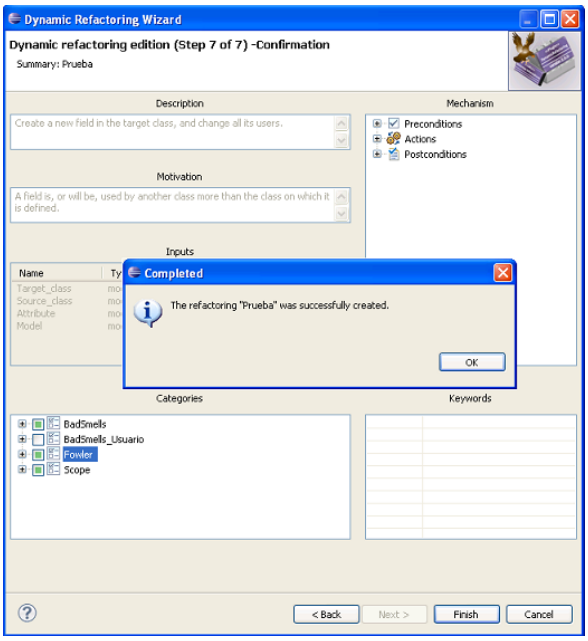

Figura 7.19: Paso 7: Resumen

```
    </categorization>
</information>
    <inputs>
            <input from="Class" method="getName" name="Old_name" root="false"
            type="moon.core.Name"/>
            <input name="Class" root="true" type="moon.core.classdef.ClassDef"/>
            <input name="New_name" root="false" type="moon.core.Name"/>
            <input name="Model" root="false" type="moon.core.Model"/>
    </inputs>
    <mechanism>
            <preconditions>
                    <precondition name="repository.moon.concretepredicate.ExistsClass">
                            <param name="Class"/>
                    </precondition>
                    <precondition name="repository.moon.concretepredicate.NotExistsClassWithName">
                            <param name="New_name"/>
                    </precondition>
            </preconditions>
            <actions>

                    <action name="repository.moon.concreteaction.RenameClass">
                            <param name="Class"/>
                            <param name="New_name"/>
                    </action>
                    <action name="repository.java.concreteaction.RenameJavaFile">
                            <param name="Class"/>
                            <param name="New_name"/>
                    </action>
                    <action name="repository.moon.concreteaction.RenameReferenceFile">
                            <param name="Class"/>
                            <param name="New_name"/>
                    </action>
                    <action name="repository.moon.concreteaction.RenameClassType">
                            <param name="Class"/>
                            <param name="New_name"/>
                    </action>
                    <action name="repository.moon.concreteaction.RenameGenericClassType">
                            <param name="Class"/>
                            <param name="New_name"/>
                    </action>
                    <action name="repository.moon.concreteaction.RenameConstructors">
                            <param name="Class"/>
```

```
                        <param name="New_name"/>
                    </action>

                </actions>
                <postconditions>
                    <postcondition name="repository.moon.concretepredicate.NotExistsClassWithName">
                        <param name="Old_name"/>
                    </postcondition>
                </postconditions>
            </mechanism>
            <examples>
                <example after="ejemplo1_despues.txt" before="ejemplo1_antes.txt"/>
            </examples>
</refactoring>
```

7.2.2. Aplicación de refactorizaciones

Una segunda fase se inicia cuando el usuario necesita ejecutar las refactorizaciones sobre código real en producción. Ahora, las piezas empiezan a trabajar conjuntamente, siguiendo el concepto de inversión de control en el *framework*, poniendo el motor en funcionamiento.

El usuario selecciona un elemento dentro del código (*e.g.* clase, atributo, método, argumento formal, etc.). En función del tipo de elemento seleccionado se presenta una vista con sólo aquellas refactorizaciones cuya entrada marcada como principal sea coincidente (ver Fig. 7.20).

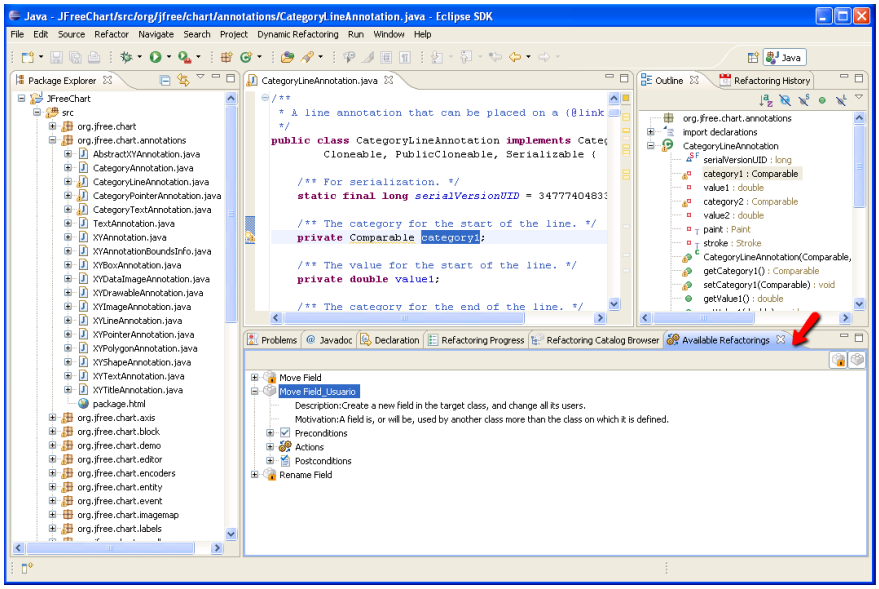

Figura 7.20: Visualización de refactorizaciones disponibles

Esto facilita la labor del usuario a la hora de aplicar refactorizaciones, puesto que sólo aquellas realmente aplicables son seleccionables, realizando un primer filtrado.

Como resultado de su selección comienza la ejecución de la refactorización. Como primer paso se muestran en un formulario las entradas adicionales. Se ejecuta la refactorización si

las precondiciones se verifican. A la finalización se verifican las postcondiciones y se recupera el código refactorizado.

Este proceso es llevado a cabo por el motor de refactorizaciones descrito en el Capítulo 6, de forma transparente para el usuario. Partiendo de la información disponible en el fichero XML que describe la refactorización y a partir de los mecanismos de reflexión que habilita el lenguaje de implementación del *plugin* (JAVA en este caso).

De hecho la aplicación de refactorizaciones incluidas en el *plugin* o incluidas en el propio entorno no se diferencian externamente en su aplicación, y consecuentemente en los efectos posteriores.

7.3. Refactorizaciones incluidas

A continuación se enumeran las refactorizaciones incluidas hasta la actualidad en el *plugin*. Sin ser un catálogo exhaustivo, se incluyen refactorizaciones en distintas categorías, según se apliquen como refactorizaciones definidas en catálogos ya existentes o definidos en este trabajo, o bien como aplicaciones en otras líneas de trabajo futuro como en la migración de bibliotecas y *frameworks*, o en el apoyo a la evolución de lenguaje.

7.3.1. Refactorizaciones básicas

Se incluyen refactorizaciones clásicas del catálogo de [Fowler, 1999] o bien existentes en la mayoría de entornos de desarrollo en JAVA.

- **ADD PARAMETER**: añade un argumento formal en la signatura de un método.
- **EXTRACT METHOD**: extrae un conjunto de instrucciones como un nuevo método. Habitualmente utilizado para evitar la repetición de código y reutilizar en su lugar el método extraído.
- **MOVE FIELD**: mueve un atributo de una clase a otra.
- **REMOVE PARAMETER**: elimina un argumento formal de la signatura de un método.
- **RENAME CLASS**: renombra una clase en el sistema.
- **RENAME FIELD**: renombra un atributo de una clase.
- **RENAME METHOD**: renombra un método en una clase.
- **RENAME PARAMETER**: renombra un argumento formal en un método.

Se han implementado renombrados a distintos niveles de granularidad (atributo, método y clase), y también operaciones de añadido y borrado sobre atributo, argumento formal o clase.

Cabe destacar, como prueba fundamental, la implementación de la refactorización **ExtractMethod**, marcada en la literatura como el Rubicón a la hora de construir herramientas de refactorización. En [Marticorena et al., 2010a] se muestran los resultados de la implementación de esta refactorización. En dicho trabajo, se describe el soporte de la refactorización por los IDEs más utilizados en Java en relación a las últimas características vinculadas a las clases y métodos genéricos. Algunos problemas detectados fueron descritos y resueltos en dicho trabajo. La solución allí aportada forma parte del soporte que se describe en esta tesis.

La implementación de una refactorización como ésta, que trabaja a nivel de instrucción, teniendo en cuenta las asignaciones entre variables, demuestra la aplicabilidad de la solución propuesta.

7.3.2. Refactorizaciones en genericidad

Ante la ausencia de refactorizaciones en genericidad en los catálogos más utilizados, en el Capítulo 5 se definió un catálogo de refactorizaciones en genericidad que se enumera a continuación:

- **Parametrizar**
- **Reemplazar parámetro formal con tipo completo**
- **Generalizar acotación**
- **Especializar acotación**

Mientras que las refactorizaciones enumeradas se adecúan a la variante de acotación por subtipado presente en Java, las refactorizaciones **Eliminar signaturas en cláusulas** y **Añadir signaturas en cláusulas**, no pueden ser incluidas dado que en Java no existe ese tipo de acotación.

7.3.3. Refactorizaciones en la migración de bibliotecas y *frameworks*

Estas refactorizaciones afrontan la evolución de un *framework* bien conocido como `JUnit` [Beck and Gamma, 1998], cambiando el código escrito para su versión 3.0 basado en herencia y convención de nombres, a código para su versión 4.0 basado en anotaciones. Los resultados parciales se presentaron en [Marticorena et al., 2008] con la implementación de dos refactorizaciones.

- **MigrateJUnit3ToJUni4TestException**: transforma tests que comprueban el lanzamiento de excepciones en el código, basado en lanzamiento y captura de excepciones con bloques `try-catch-finally` y métodos `fail()`, simplificando el código al uso de anotaciones `@Test(expected=NombreExcepción.class)`.

- **MigrateJUnit3ToJUnit4**: transforma aquellos tests que siguen la convención de nombres (su nombre comienza por `test`, añadiendo la anotación marcadadora `@Test`.

El objetivo era doble, por un lado validar la solución propuesta para la aplicación de transformaciones que permiten la evolución entre diferentes versiones de *frameworks* ampliamente utilizados. En segundo lugar validar la adecuación de JavaMOON para dar soporte a refactorizaciones sobre una característica no incluida incialmente en MOON, como son las anotaciones en Java, pero incluida mediante herencia como un componente más del metamodelo JavaMOON.

7.3.4. Refactorizaciones en la evolución del lenguaje

Este conjunto de transformaciones surgen de la evolución del lenguaje de programación. En concreto de la evolución de Java entre diferentes versiones al incluir la anotación @Override en la redefinición de métodos y el uso de tipos enumerados en lugar de constantes. Mientras que la refactorización para el uso de tipos enumerados todavía está en una primera fase de desarrollo, en el segundo caso se obtuvieron resultados comparables a los producidos por entornos como Eclipse tal y como se describe en [Marticorena et al., 2011b]. En dicho trabajo, también se incluyó **Parametrizar** como refactorización vinculada a la evolución del lenguaje, dada la inclusión tardía en Java de la genericidad.

- **Enumerated Types**: se transforma el código que simula la definición de tipos enumerados con constantes en interfaces, siendo sustituidos por el uso de tipos enumerados incluidos en Java 1.5. Aunque se trata de una primera versión, permite validar el soporte de los tipos enumerados en el metamodelo JavaMOON y su inclusión en refactorizaciones.
- **Add Override Annotation**: se añaden anotaciones `@Override` en aquellos métodos que redefinen a métodos en los ancestros de la jerarquía de herencia.

7.4. Regeneración del código fuente

Una vez ejecutadas las refactorizaciones a través del prototipo implementado, se debe recuperar el código Java refactorizado. Tomando como base la solución basada en el patrón de diseño **Visitor** y **Strategy** [Gamma et al., 1995] descrita en la Sec. 6.4 (ver Fig. 6.5), se muestra en la Fig. 7.21 el diagrama de clases correspondiente para la regeneración de código Java.

Se describe a continuación el rol de cada una de las clases, dentro de dicho diagrama:

`JavaSourceCode` representa el conjunto de líneas de código Java recuperado a partir de las instancias del metamodelo JavaMOON. Un objeto `SourceCode` es una representación de ficheros de texto compuestos de líneas, en este caso particular, su descendiente

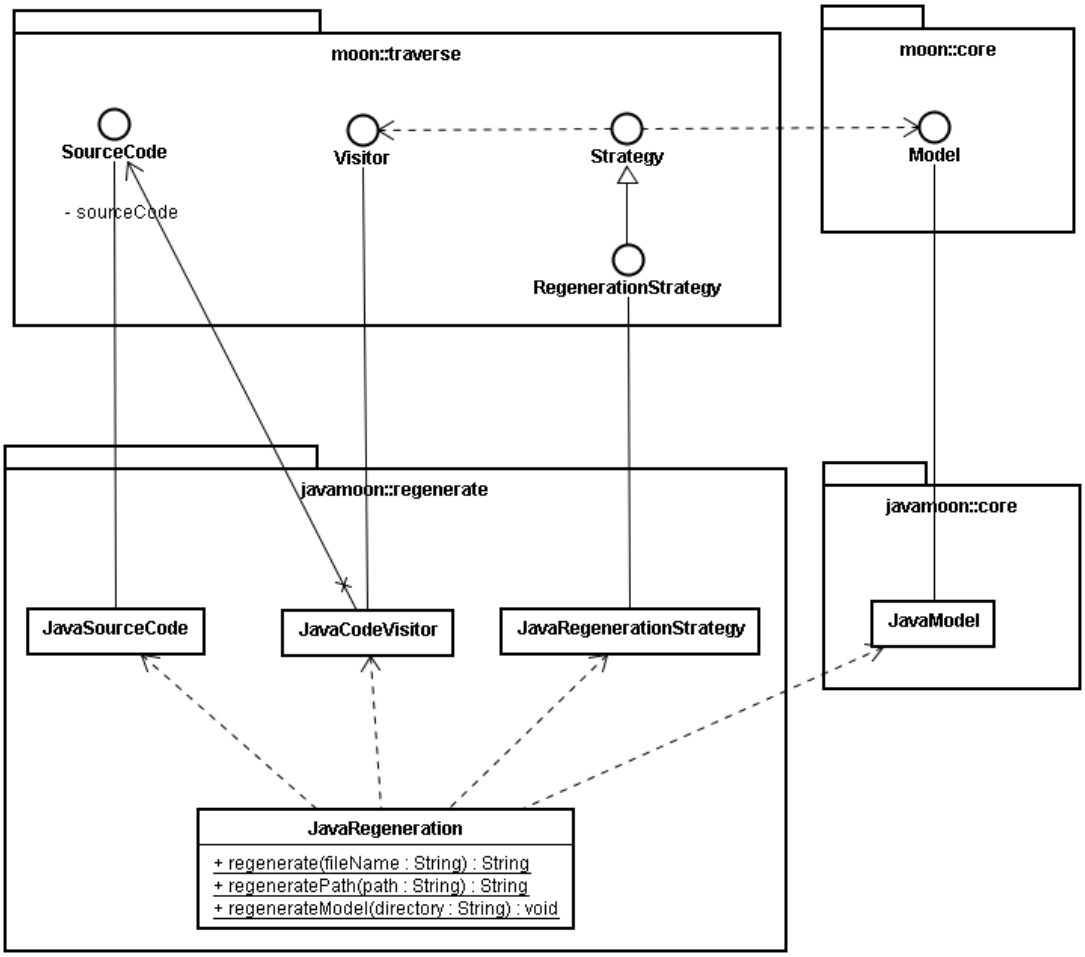

Figura 7.21: Regeneración de código JAVA

se corresponderá físicamente con un fichero con extensión `.java`, y podrá contener el código fuente de una o varias clases.

`JavaVisitor` realiza las visitas sobre los elementos que conforman el metamodelo JAVAMOON. Existe una correspondencia uno a uno entre los elementos fundamentales del metamodelo (*e.g.* `NameSpace`, `ClassDef`, `ClassType`, `Attribute`, `Method`, etc.) y el correspondiente método de visita, en la interfaz `Visitor`. Las particularidades del lenguaje JAVA se incluyen en la clase `JavaVisitor` a través de métodos privados. En función de la estrategia elegida, en este caso la regeneración de código, completa línea a línea el texto a partir de la información del elemento visitado, completando la información de un objeto `JavaSourceCode`.

JavaRegenerationStrategy define la estrategia de recuperación de código a partir del `Visitor` y `Model` establecido, en este caso instancias concretas de `JavaVisitor` y `JavaModel`.

JavaModel objeto con toda la información del código. Inicialmente contiene la información extraída a partir de los ficheros fuente y completada con la información binaria de las bibliotecas necesarias para una correcta compilación del código. El estado puede variar posteriormente al haberse aplicado operaciones de refactorización a través del motor. El proceso de regeneración de código es independiente de que el modelo haya sufrido o no alguna transformación.

JavaRegeneration coordina las operaciones de regeneración, inyectando al objeto `JavaRegenerationStrategy` los correspondientes `Visitor`, `Model` y `SourceCode` para la regeneración del código. En este caso, las correspondientes instancias concretas de `JavaVisitor`, `JavaModel` y `JavaSourceCode`, al trabajar sobre código JAVA.

Como ya se indicó en el Capítulo 6, la recuperación del código sufre de una alta dependencia del lenguaje objetivo de la refactorización. En concreto, aunque en todos los lenguajes aparece el concepto de código fuente (en nuestro caso el concepto de `SourceCode`), la estructura física de directorios y ficheros en la que se ve reflejada esa información, es muy dependiente del lenguaje objetivo.

En el caso de JAVA, nos encontramos con la particularidad de la correspondencia uno a uno del concepto lógico de paquete/subpaquete con el concepto físico de directorio. Por lo tanto, cada espacio de nombres (`NameSpace`) en MOON, se reflejará como `JavaPackage` en JAVAMOON, y finalmente como directorios o subdirectorios físicos en el sistema de ficheros, según el anidamiento.

Por otro lado, el fichero fuente en JAVA coincide en nombre con la clase pública declarada en dicho fichero (sólo una clase pública por fichero). Aunque en el mismo fichero pueden existir otras clases, no podrán tener dicho modificador. Este contraste entre un elemento físico (el fichero fuente `.java`) y el elemento lógico (clases en el modelo lógico) se complica más dado que en JAVA existen los conceptos de clases internas, locales y anónimas.

Concluyendo, todas estas particularidades, no generalizables a otros lenguajes objetivo, motivan que el módulo regenerador se vincule de manera concreta al lenguaje objetivo, siendo imposible una generalización aplicable a un conjunto amplio de lenguajes. En nuestra solución propuesta, es la clase `JavaRegeneration` la que centraliza las particularidades del ajuste entre la estructura lógica contenida en la instancia del metamodelo JAVAMOON, y la correspondiente estructura física a la hora de recuperar el conjunto de directorios/ficheros con el código fuente.

7.5. Análisis o matriz DAFO del prototipo.

En esta sección se concluye el capítulo realizando un análisis de debilidades, amenazas, fortalezas y oportunidades del prototipo. Como resultado de la propia evolución en el desarrollo

del *framework* de refactorización, aplicado a un lenguaje objetivo concreto como JAVA, y su inclusión como *plugin* en un entorno IDE bien conocido como ECLIPSE, se han podido detectar las características que se presentan a continuación en forma de matriz DAFO.

Tabla 7.1: Matriz DAFO del prototipo

Debilidades	**Amenazas**
Necesidad de un alto conocimiento por parte del analista de refactorizaciones de los repositorios de predicados, funciones y acciones.	Necesidad de herramientas de parseo del código adecuadas para la generación de los metamodelos del lenguaje objetivo.
Requiere un alto conocimiento del lenguaje objetivo y alto esfuerzo para la construcción del metamodelo asociado.	Generación del metamodelo objetivo en el lenguaje de implementación seleccionado.
Alto esfuerzo en la definición y construcción de las refactorizaciones iniciales.	Cambios futuros en el lenguaje objetivo pueden originar cambios o ampliaciones en su metamodelo.
Problemas de rendimiento y escalabilidad en relación a los ficheros fuente y binarios a procesar.	
Fortalezas	**Oportunidades**
Construcción con reutilización de las refactorizaciones, simplificando el proceso a medida que los repositorios se completan.	Aplicación a nuevos catálogos de refactorizaciones.
Fácil ampliación y modificación de las refactorizaciones en caliente.	Aplicación de la solución planteada a la migración de bibliotecas y *frameworks*.
Integración de mecanismos de indexación y consulta de las refactorizaciones facilitando su uso por parte del usuario.	Aplicación en la construcción de motores para la modificación de código ante la evolución de los lenguajes de programación.

Como se ha señalado, desde el punto de vista de las oportunidades de aplicación que ofrece esta solución, existe un conjunto amplio, no limitado en particular a la definición y ejecución de refactorizaciones, sino a otras líneas de investigación, como se detallará en el Capítulo 8 en su Sec. 8.4.

CAPÍTULO 8

CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO

Este capítulo aborda las conclusiones obtenidas en el desarrollo del presente trabajo. Partiendo de los objetivos establecidos inicialmente en el Capítulo 1, se extraen aquellas conclusiones más importantes, en base a los resultados obtenidos.

Dicha exposición se realiza desde un enfoque general en la Sec. 8.1, y con un estudio más detallado en la Sec. 8.2, para finalmente, en la Sec. 8.3, exponer algunas debilidades y limitaciones presentes en la propuesta actual.

Por otro lado, a lo largo de este proceso de investigación, se han ido obteniendo ciertos resultados que abren nuevas vías de futuro en la investigación sobre la refactorización del software. Dado lo amplio del contexto inicial, partiendo de la evolución y mantenimiento del software, y entendiendo las refactorizaciones como transformaciones de reescritura del código (ver Capítulo 2), se establece un amplio abanico de nuevas líneas de trabajo futuro. Estas líneas están descritas en la Sec. 8.4.

8.1. Conclusiones generales

Recordando lo planteado en el Capítulo 1, se estableció como objetivo fundamental de esta tesis:

> ***La definición y ejecución de refactorizaciones con el mayor grado de reutilización para la familia de lenguajes orientados a objetos, estática o fuertemente tipados, con capacidades para programación genérica.***

Partiendo de la situación inicial descrita en el estado del arte (ver Capítulo 2) y a la luz de los resultados obtenidos, se ha conseguido formalizar en un mayor grado la definición de refactorizaciones. Con el fin de eliminar ese cierto grado de subjetividad y falta de precisión, se ha establecido una estructura más precisa para su definición (ver Capítulo 4).

Por otro lado, el metamodelo MOON, detallado en el Capítulo 3, permite representar un amplio conjunto de propiedades y características comunes en la familia de LPOO de la corriente principal, proporcionando una sólida base para las validación de las refactorizaciones definidas y construidas sobre dicho metamodelo. El empleo del lenguaje ALF permite dicha validación, garantizando un grado de corrección que en otras definiciones de la literatura de refactorizaciones no existía. Se evita cualquier subjetividad, quedando la refactorización claramente determinada.

El metamodelo MOON contiene las características fundamentales dentro de la programación genérica. Así pues, la definición de un catálogo de refactorizaciones centradas en genericidad sobre dicho metamodelo (ver Capítulo 5) utilizando ALF como lenguaje, permite constrastar la validez de MOON en dicho campo, frente a la ausencia de otros trabajos en esta línea.

Para la posterior implementación y ejecución de las refactorizaciones, se ha presentado una base sólida y reutilizable, a partir de una solución basada en *frameworks*. El motor de refactorizaciones descrito (ver Capítulo 6), y la facilidad para conectar y extender los elementos de los repositorios, tanto en MOON, como con la extensión concreta a un lenguaje de la corriente principal como JAVA (ver Capítulo 7), aportan una solución reutilizable que debería ser aplicable en el contexto de otros lenguajes similares.

Finalmente, se puede concluir que el objetivo general de la tesis ha sido completado, siendo conscientes de aquellas limitaciones y debilidades de la propuesta, como se detalla en la Sec. 8.3.

8.2. Conclusiones desglosadas por objetivos

Se analizan a continuación cada uno de los sub-objetivos básicos establecidos en la Sec. 1.7, revisando el grado de cumplimiento de cada punto en particular, y describiendo las aportaciones realizadas en esta tesis doctoral.

Almacenamiento y recuperación de la información de código

En concreto para lenguajes OO estáticamente tipados, utilizando MOON como solución de base para su reutilización.

El soporte de los LPOO estáticamente tipados ha sido abordado desde la independencia del lenguaje en el Capítulo 3, presentando una solución basada en el metamodelo MOON. La solución aportada cubre la amplia mayoría de características de dichos lenguajes, permitiendo posteriormente su adecuación a los lenguajes de la corriente principal. Se aborda el soporte

de variaciones a través de *hot-spots* en el *framework* y de inversión de control en el Capítulo 6, y se aplica la solución a casos concretos sobre un lenguaje como JAVA en el Capítulo 7.

Caracterización de las refactorizaciones en base a los elementos que la conforman

Incluyendo un modelo de caracterización que permita clasificar y establecer relaciones de cara a la implementación y reutilización de las mismas. Desde el punto de vista del constructor de herramientas y del programador como usuario de la herramienta.

En el Capítulo 4, Sec. 4.2, se ha especificado una caracterización de la refactorizaciones que cubre los objetivos planteados. A partir de la definición de la refactorización, se caracteriza de forma que se pueda dar respuesta a cuestiones habituales en el desarrollo de herramientas de refactorización. Entre ellas, el orden de implementación, reutilización en la creación de la interfaz gráfica asociada, nivel de interacción con el usuario, etc.

Definición de los elementos básicos de las refactorizaciones

Se tomará MOON como marco de trabajo, identificando los puntos de variabilidad necesarios para la aplicación posterior de refactorizaciones sobre lenguajes concretos.

En el Capítulo 4, partiendo del metamodelo MOON detallado en el Capítulo 3, se establece la estructura para la definición de una refactorización, mejorando estructuras previas utilizadas en otros catálogos. Los elementos que componen esta estructura serán especificados mediante el lenguaje ALF e implementados posteriormente en repositorios, tal y como se concluye en el Capítulo 6, para dar una solución extensible a otros lenguajes.

Catálogo de refactorizaciones en programación genérica

Se plantea un nuevo conjunto de refactorizaciones centradas en cuestiones relativas a la programación genérica a través de las clases y métodos genéricos soportados en los lenguajes de programación orientados a objetos.

En el Capítulo 5 se ha incluido un catálogo de refactorizaciones sobre programación genérica. La aportación es doble. Por un lado se ha incluido un nuevo catálogo no incluido en anteriores trabajos, tomando como foco las clases genéricas. Por otro lado, se ha validado el soporte de MOON a las propiedades particulares en genericidad y la utilización de un lenguaje como ALF para la especificación de dichas refactorizaciones sobre MOON, pero aplicable a otros metamodelos.

Validación de la solución planteada sobre el catálogo de refactorizaciones

Se revisa la solución planteada en la construcción de las refactorizaciones del nuevo catálogo establecido previamente, validando la solución propuesta.

En el Capítulo 7, se plantea el caso de estudio sobre el lenguaje JAVA, desde el soporte a la información extraída del código escrito en este lenguaje objetivo, la implementación de

funciones, predicados y acciones, y la creación final de un prototipo que integre las refactorizaciones definidas.

Establecer un proceso de refactorización

Asegurando en cierta medida la preservación del comportamiento, como resultado de la aplicación de las refactorizaciones definidas.

En el Capítulo 4, en la Sec. 4.3, se define un proceso de definición y construcción de refactorizaciones, basado en la estructura de definición de refactorizaciones establecida en el Capítulo 4. Partiendo de una definición de roles, tareas y productos, se establece el proceso en el que participan todos estos elementos, utilizando SPEM para su definición. Finalmente se propone un proceso iterativo e incremental que garantiza una cierta preservación del comportamiento al aplicarse las refactorizaciones.

Construcción de un prototipo

Donde se incluyan los elementos previamente definidos.

Se concluye con la integración del prototipo como *plugin* del entorno Eclipse. El *plugin* permite construir y ejecutar las refactorizaciones con un menor esfuerzo, apoyándose en la infraestructura presentada en el Capítulo 6.

8.3. Limitaciones y debilidades

Pese a los resultados obtenidos, somos conscientes de ciertas limitaciones de la propuesta actual, desde una doble visión.

Desde el punto de vista de la independencia del lenguaje, es imposible dar una solución completamente independiente del lenguaje en refactorización, como se señaló en el Capítulo 3. La propia naturaleza del concepto de refactorización, aplicada sobre un lenguaje concreto, motiva que la refactorización esté influenciada por las características particulares del lenguaje objetivo. Dada la variedad de características diferentes, incluso entre lenguajes de una misma familia, junto con la imposibilidad de predecir nuevas características que se van incorporando a los lenguajes en su evolución, imposibilita el poder dar una solución definitiva a dicho problema.

Sin embargo, en este trabajo se ha dado una solución que permite un alto grado de reutilización a la hora de abordar la misma refactorización para diferentes LPOO. Las refactorizaciones no son construidas desde cero, sino que a partir del metamodelo MOON, el conjunto de predicados, funciones y acciones definidos sobre dicho metamodelo y el motor de refactorizaciones, se facilita la definición, construcción y ejecución de las refactorizaciones para el lenguaje objetivo concreto.

Las mayores dificultades se encuentran en los procesos de extracción y regeneración del código, condicionados al lenguaje de implementación del prototipo o herramienta construida.

Esta tarea está supeditada a la posible existencia y reutilización de bibliotecas o herramientas que faciliten el análisis y recorrido del código. Otros problemas, que no se tenían identificados desde el inicio, como la necesidad de obtener información adicional sólo disponible en ficheros binarios, y el volumen de carga de la información a manejar, dificultan el desarrollo y el rendimiento final en las implementaciones realizadas.

El otro punto de vista desde el que se observan limitaciones es desde la preservación del comportamiento. En la mayoría de trabajos del estado del arte, este punto se ha abordado de manera similar a nuestra aproximación. La corrección de la refactorización sigue descansando sobre la base de un juego completo y correcto de precondiciones. La ausencia de tests o baterías de pruebas en los sistemas software a refactorizar, dificulta la comprobación de la corrección de las mismas una vez aplicadas, y por tanto impide verificar finalmente la adecuación de las precondiciones establecidas.

Pero, aceptando ciertas limitaciones en la preservación del comportamiento en nuestro enfoque, se ha mejorado la garantía del grado de corrección de la refactorización (comprobación sintáctica y de semántica estática) frente a otras propuestas. En tanto en cuanto no se proporcionen herramientas de comprobación dinámica para el lenguaje ALF, la preservación del comportamiento debe seguir descansando en la ejecución de tests automáticos, siempre y cuando estos no sean también objetivo de la refactorización, problema no abordado en el presente trabajo.

A pesar de las limitaciones detectadas, a nuestro juicio se ha mejorado en el soporte a las refactorizaciones con un cierto grado de independencia del lenguaje, siempre desde un enfoque basado en reutilización, y con unas ciertas garantías en cuanto a la preservación de comportamiento.

8.4. Líneas de trabajo futuro

Como es habitual en todo trabajo de investigación, se han detectado un gran número de líneas de trabajo que pueden ser abordadas desde la perspectiva de la refactorización del software en el futuro.

Frente al enfoque habitual de aplicación de refactorizaciones en labores de mantenimiento, surgen varias posibilidades relativas a la evolución de los lenguajes y plataformas de desarrollo. La evolución de los lenguajes objetivo de la refactorización, y los cambios de diseño a que ésta conduce sobre bibliotecas y *frameworks*, hace que sea necesario un estudio en profundidad de la aplicación de las refactorizaciones en dichas líneas.

8.4.1. Aplicación de refactorizaciones definidas con MOON a otros lenguajes

Como se estableció en el Capítulo 3, los LPOO tienen un conjunto de propiedades comunes que hacen que un planteamiento como el aquí presentado sea adecuado para abordar el

problema de lograr cierta independencia del lenguaje. Dentro de las líneas de trabajo futuro, se establece la extensión de MOON a otros lenguajes para la aplicación de refactorizaciones.

MOON surgió como notación minimal y se validó inicialmente sobre Eiffel [Crespo, 2000], pero queda abierta su aplicación sobre lenguajes que han ocupado en la última década los primeros puestos en estadísticas de uso como por ejemplo C#. Dada su similitud con Java (debido a su origen), hace previsible que la construcción de la extensión equivalente para C# sea abordable. Aunque por otro lado, el conjunto de propiedades adicionales que incluye el lenguaje, y una cierta desviación en sus últimas versiones respecto a la especificación de Java, hacen intuir que la extensión particular incluirá un conjunto amplio de nuevas clases.

Por otro lado, como solución formal a la hora de especificar las refactorizaciones, se ha utilizando un lenguaje de acciones Alf, que a su vez mantiene muchas de las propiedades comunes a los LPOO: paquetes, tipos, clases, expresiones, instrucciones, etc. El volumen de código escrito en dicho lenguaje crece a medida que se quiere dar un soporte formal a la definición de refactorizaciones sobre el metamodelo. Queda pues abierta la posibilidad de que a medida que el uso de Alf se extienda, y no sólo en el contexto de este trabajo, surgirá el problema de la evolución y mantenimiento de software escrito con dicho lenguaje. Ante esta situación, parece previsible que se tengan que realizar refactorizaciones sobre ficheros Alf, bien de manera independiente, o bien sincronizados con el código generado.

Por lo tanto, queda abierta la posibilidad de utilizar MOON como metamodelo a partir del cual realizar una extensión de un lenguaje como Alf, con el fin de dar una solución de soporte a su refactorización, con cierto grado de independencia y reutilizando los esfuerzos previos. Refactorizaciones como **Rename Class** o **Extract Method**, estudiadas en trabajos sobre MOON, son extrapolables a su aplicación sobre Alf, y su inclusión futura en herramientas de desarrollo y/o refactorización.

8.4.2. Refactorización en la migración del software

La evolución del software se ve sujeta a factores como los cambios de los lenguajes de programación en los que finalmente se implementa la solución de diseño. Es habitual, que productos implementados en un cierto lenguaje cambien parte de su estructura interna en base a nuevas posibilidades que se han incluido en versiones posteriores de la especificación del lenguaje. Un gran número de estas modificaciones entran dentro de lo que habitualmente se denomina como *"azúcar sintáctico"*, y aunque en algunos casos estas nuevas características no incorporan grandes mejoras, en otros casos incrementan la facilidad de desarrollo y mantenimiento, aumentando la calidad del software.

En la mayoría de los casos, los cambios a los que conducen se pueden clasificar, siguiendo la taxonomía de [Visser, 2001], como reescritura de programas: transformaciones que cambian un programa dentro del mismo lenguaje (el lenguaje origen y destino son el mismo). Normalmente este proceso de reescritura conlleva algún resultado, como mejoras en el diseño, mayor facilidad de comprensión, etc.

Estos cambios, sin ser fundamentales para el correcto funcionamiento, entran dentro de la lógica evolución del sistema software. Muchas de las características que se reescriben están

en la categoría de propiedades en desuso[1], que no tienen garantizado un mantenimiento en el futuro.

Tomando como ejemplo el funcionamiento del *framework* JUnit, sus versiones previas se basaban en mecanismos de herencia y convención de nombres a la hora de escribir el código de los tests. Posteriormente el *framework* resolvía a través de mecanismos de reflexión la recolección de los tests a ejecutar. En las siguientes versiones (versión 4 en adelante) se aprovecharon las nuevas mejoras del lenguaje. La evolución sigue la línea de un *framework* de caja gris, se elimina el mecanismo de herencia y se evoluciona hacia mecanismos declarativos (anotaciones) para marcar el rol de los métodos en las clases e importaciones estáticas. Todo el código escrito para el *framework* JUnit 3 puede ser reescrito para funcionar en la nueva versión. La transformación ejerce una enorme influencia en el *framework* equivalente en la plataforma .NET, denominado NUnit, siguiendo el mismo diseño basado en atributos[2]. Aunque la transformación de una versión a otra pueda parecer trivial, no por ello deja de ser una labor pesada y propensa a fallos si se realiza manualmente y sin comprobaciones adicionales.

El trabajo desarrollado en esta tesis doctoral, fue también aplicado a la migración entre versiones de este *framework*. Mediante un *plug-in* desarrollado para Eclipse, se validó la construcción y ejecución de las refactorizaciones que permitían la migración de tests escritos para JUnit 3 a su versión 4 [Marticorena et al., 2008]. En concreto se implementaron dos refactorizaciones: **MigrateJUnit3ToJUnit4** para cambiar los tests con asertos y **MigrateJUnit3ToJUni4TestException** para cambiar los tests que comprueban el lanzamiento de excepciones.

Dados los resultados iniciales, queda abierta una línea de definición de catálogos de refactorizaciones para facilitar la migración entre distintas versiones de *frameworks*, no sólo limitándonos a este ejemplo concreto. Dicha línea diverge de la línea general planteada en esta tesis, dado que se restringe a un lenguaje particular, pero por otro lado los resultados obtenidos parecen aplicables a este tipo de operaciones de migración, incluyendo su integración y soporte en entorno de desarrollo. A nuestro juicio, la posibilidad de disponer de herramientas que faciliten estas migraciones es de gran interés para la comunidad de desarrolladores.

8.4.3. Refactorización del software y evolución de los lenguajes de programación

La evolución de los lenguajes de programación suele implicar la inclusión de nuevos elementos en el lenguaje o en sus bibliotecas. Dichos cambios llevan a que los patrones, *idioms* y técnicas utilizadas anteriormente puedan ser replanteados a partir de las nuevas características introducidas. Sin embargo el cambio manual suele ser propenso a errores, consume mucho tiempo y es caro de llevar a cabo desde un punto de vista económico.

Este cambio se ha denominado en la literatura como *"rejuvenecer el código fuente"* [Pirkelbauer et al., 2010], definido como *"la transformación fuente a fuente que reemplaza propie-*

[1] Traducción libre del término *deprecated* habitualmente utilizado.
[2] En .NET se utiliza el término atributo en lugar de anotación.

dades del lenguaje en desuso por nuevas soluciones". Entre sus objetivos se vuelve a plantear la reducción de la entropía del software, punto en común con la refactorización. Así pues, la refactorización podría aplicarse también como solución al problema del rejuvenecimiento del código.

Pero se debe señalar que mientras la refactorización marca como objetivo básico la mejora de la estructura del código y su mantenimiento, el rejuvenecimiento del código fuente tiene otros objetivos [Pirkelbauer et al., 2010]:

- Migración del código fuente de versiones previas a actuales del lenguaje elegido.
- Educación de los programadores, con la inclusión de sugerencias de cambio y migración en los IDE actuales.
- Optimización del código.

Además, las refactorizaciones están vinculadas a la POO, mientras que el concepto de rejuvenecimiento del código es más amplio, sin estar ligado a un paradigma concreto. La Tabla 8.1 muestra la comparativa entre ambos conceptos realizada en [Pirkelbauer et al., 2010].

Tabla 8.1: Comparativa entre rejuvenecimiento y refactorización (tomada de [Pirkelbauer et al., 2010])

	Rejuvenecimiento de código	Refactorización
Transformación	Fuente a fuente	Fuente a fuente
Preservación del comportamiento	Mejora del comportamiento	Preservación del comportamiento
Dirección	Sí (Eleva el nivel de abstracción)	No
Drivers	**Evolución del lenguaje y bibliotecas**	**Cambios de diseño**
Indicadores	Técnicas (*Workaround techniques*) / *idioms*	Defectos de código y antipatrones
Aplicaciones	Una vez en la migración del código fuente	Tareas de mantenimiento recurrentes

La diferencia fundamental es que, en el rejuvenecimiento de código, los cambios están dirigidos por la evolución del lenguaje de programación, no por cambios en el diseño. Así pues el rejuvenecimiento de código parece ligado al lenguaje particular sobre el que se ha realizado el cambio, dificultando en mayor medida la independencia del lenguaje de programación, objetivo del presente trabajo. Sin embargo, la solución propuesta, permite dar soporte también a este tipo de operaciones.

Es discutible si existe una relación de subconjuntos entre el rejuvenecimiento del código y la refactorización (o viceversa). Pero parece obvio que si se dispone de una herramienta que permita dicha migración puede aplicarse a un gran número de programas escritos en la misma versión del lenguaje [Pirkelbauer et al., 2010], sin mayor necesidad de interacción con el usuario como ocurre por ejemplo en las refactorizaciones. Desde ese punto de vista, el proceso puede ser automatizable.

En [Overbey and Johnson, 2009], se propone la utilización de herramientas de refactorización para resolver estas tareas. Desde su punto de vista, la adición continua de nuevas propiedades a los lenguajes hace que aparezcan una diversidad de "dialectos", planteándose un problema muy grave: *"el código no evoluciona con el lenguaje"*. Si las herramientas de refactorización pueden dar una solución de soporte al cambio en aplicaciones, bibliotecas y *frameworks*, también podrían *"reemplazar propiedades del lenguaje caducadas por su equivalentes actuales"*.

Su propuesta va más allá, proponiendo que en las publicaciones de nuevas versiones de los lenguajes, se acompañe del conjunto de refactorizaciones que migran el código de la versión previa a la actual. Aunque los autores no emplean el termino de "rejuvenecimiento", sino refactorizaciones, se pueden establecer múltiples coincidencias con lo establecido en [Pirkelbauer et al., 2010]. Desde este punto de vista la herramienta de refactorización (y rejuvenecimiento) pasaría a ser un elemento más del paquete de desarrollo básico a distribuir, junto con compiladores, depuradores, decompiladores, etc.

Esta visión no hace sino ampliar la necesidad de herramientas de refactorización integradas, bien como parte de los entornos de desarrollo integrados, de los *kit* de desarrollo o de otras herramientas adicionales que se usen en el desarrollo y mantenimiento del software.

Para validar la posibilidad de reutilizar los resultados obtenidos, en [Marticorena et al., 2011b] se afronta la evolución de los lenguajes a través de refactorizaciones. De nuevo, aplicando la arquitectura propuesta se valida definiendo, construyendo y aplicando refactorizaciones que renuevan el código envejecido, añadiendo propiedades incluidas en las nuevas versiones del lenguaje JAVA.

La línea de trabajo abierta por distintos autores en cuanto a la automatización de esa renovación del código, parece que puede ser completada a través de herramientas de refactorización. Sin embargo, debe realizarse un trabajo más profundo en esta línea.

Apéndices

APÉNDICE A

SINTAXIS CONCRETA DE MOON

La gramática que se presenta a continuación define la sintaxis concreta de MOON y por tanto determina la estructura de las clases que se pueden definir. Una descripción más completa de la gramática, su sintaxis abstracta asociada, reglas del sistema de tipos y simplificaciones, se encuentra disponible en [Crespo, 2000, Capítulo 4].

La gramática MOON se especifica mediante una variante de LDL (Language Description Language) [ISE, 1992]. Los elementos del lenguaje de descripción se pueden resumir brevemente como:

- Las comillas (‘’) delimitan los literales. Los literales que son palabras claves del lenguaje se escriben en negrita para enfatizar en ellos. Por ejemplo **class** indica que es un literal que representa una palabra clave.
- Para los símbolos no terminales se utilizan identificadores en mayúsculas. Por ejemplo: `HEADER` indica que `HEADER` es un no terminal.
- `{const sep ...}+` indica que la construcción const se repite una o más veces, donde cada ocurrencia se separa de otra con `sep`. Si se omite el símbolo + significa que la repetición se realiza cero o más veces.
- `[CONSTR]` indica que la construcción `CONSTR` es opcional.
- | separa alternativas.
- Para los símbolos terminales se utilizan identificadores en minúsculas. Por ejemplo, `dígitos` indica que `dígitos` es un terminal. En la gramática que aquí se presenta no se llegan a definir los símbolos terminales ni los terminales indirectos. Los símbolos terminales indirectos son símbolos no terminales cuya regla de definición está dada por un terminal.

El símbolo no terminal MODULE es el símbolo de partida. Aparecen resaltadas algunas partes de la gramática, indicando dónde se encuentran representadas las relaciones de herencia y cliente que introduce la clase y con qué clases se relaciona. También se señalan las reglas donde se aprecian las simplificaciones asumidas en MOON y las construcciones que dan lugar a las variantes de acotación del modelo.

```
1   MODULE              ≜ CLASS_DEF
2   CLASS_DEF           ≜ [deferred] class CLASS_NAME HEADER SIGNATURES CLASS_BODY end
3   CLASS_NAME          ≜ CLASS_ID [FORMAL_PARAMETERS]
4   HEADER              ≜ INHERITANCE_LIST
5   SIGNATURES          ≜ signatures SIG_LIST
6   CLASS_BODY          ≜ body { METHODS_IMPL ';' ...}
7   FORMAL_PARAMETERS   ≜ '[' { FORMAL_PAR ',' ...}+ ']' [ BOUND_W ]          variante w
8   FORMAL_PAR          ≜ FORMAL_GEN_ID [ BOUND_S ]                           variante s
9   INHERITANCE_LIST    ≜ { INHERITANCE_CLAUSE ';' ...}
10  INHERITANCE_CLAUSE  ≜ inherit CLASS_TYPE OPLUS                            hereda
11  OPLUS               ≜ { MODIFIER ';' ...}
12  MODIFIER            ≜ rename PROP_ID as PROP_ID | redefine PROP_ID | makedeferred PROP_ID |
                          makeeffective PROP_ID | select PROP_ID
13  SIG_LIST            ≜ ATTRIBUTE_DECS METHOD_DECS
14  ATTRIBUTE_DECS      ≜ attributes { ATT_DEC ';' ...}
15  METHOD_DECS         ≜ methods { METH_DEC ';' ...}
16  ATT_DEC             ≜ VAR_DEC
17  METH_DEC            ≜ ROUTINE_DEC | FUNCTION_DEC
18  ROUTINE_DEC         ≜ WITHOUT_RESULT
19  FUNCTION_DEC        ≜ WITHOUT_RESULT ':' TYPE                             es cliente
20  WITHOUT_RESULT      ≜ [deferred] METHOD_ID [FORMAL_ARGUMENTS]
21  FORMAL_ARGUMENTS    ≜ '(' { VAR_DEC ',' ...}+ ')'
22  VAR_DEC             ≜ VAR_ID ':' TYPE                                     es cliente
23  TYPE                ≜ FORMAL_GEN_ID | CLASS_TYPE
24  CLASS_TYPE          ≜ CLASS_ID [REAL_PARAMETERS]                          ¿de qué clases?
25  REAL_PARAMETERS     ≜ '[' { TYPE ',' ...}+ ']'                            ¿de qué clases?
26  METHOD_IMPL         ≜ NONE_DEFERRED_R | NONE_DEFERRED_F
27  NONE_DEFERRED_R     ≜ MSIG [LOCAL_DECS] METHOD_BODY
28  NONE_DEFERRED_F     ≜ MSIG ':' TYPE [LOCAL_DECS] METHOD_BODY
29  LOCAL_DECS          ≜ { VAR_DEC ';' ...}+
30  METHOD_BODY         ≜ do INSTR end
31  MSIG                ≜ METHOD_ID [FORMAL_ARGUMENTS]
32  INSTR               ≜ COMPOUND_INSTR | CREATION_INSTR |
                          ASSIGNMENT_INSTR | CALL_INSTR          no hay instrucciones de control
33  COMPOUND_INSTR      ≜ {INSTR ';' ...}
34  CREATION_INSTR      ≜ create VAR_ID
```

35	ASSIGNMENT_INSTR	≙ VAR_ID ':=' EXPR	
36	CALL_INSTR	≙ CALL_INSTR_LONG1 | CALL_INSTR_LONG2	*no hay llamadas en cascada*
37	CALL_INSTR_LONG1	≙ METHOD_ID [REAL_ARGUMENTS]	
38	CALL_INSTR_LONG2	≙ CALL_EXPR_LONG1'.'CALL_INSTR_LONG1	
39	EXPR	≙ MANIFEST_CONSTANT | CALL_EXPR	*se reducen las formas de expresiones*
40	MANIFEST_CONSTANT	≙ **nil** | REAL_CONSTANT | INTEGER_CONSTANT | BOOLEAN_CONSTANT | CHAR_CONSTANT | STRING_CONSTANT	
41	CALL_EXPR	≙ CALL_EXPR_LONG1 | CALL_EXPR_LONG2	*no hay llamadas en cascada*
42	CALL_EXPR_LONG1	≙ ENTITY [REAL_ARGUMENTS]	
43	CALL_EXPR_LONG2	≙ ENTITY'.'CALL_EXPR_LONG1	
44	REAL_ARGUMENTS	≙ '(' {EXPR_ATOM ',' ...}+ ')'	
45	EXPR_ATOM	≙ MANIFEST_CONSTANT | CALL_EXPR_LONG1	
46	ENTITY	≙ VAR_ID | **result** | **self**	
47	PROP_ID	≙ VAR_ID | METHOD_ID	

Los símbolos BOUND_S y BOUND_W que aparecen en las reglas anteriores, dependen de qué variante de acotación de parámetros genéricos formales esté en consideración. Las reglas correspondientes a estos símbolos se definen a continuación de acuerdo a la variante de lenguaje a la que se dará lugar.

Reglas de variantes:

48S	BOUND_S	≙ '->'CLASS_TYPE	*es cliente*
48W	BOUND_S	≙ ϵ	
49S	BOUND_W	≙ ϵ	
49W	BOUND_W	≙ **where** { WHERE_CLAUSE ',' ...}+	
50W	WHERE_CLAUSE	≙ FORMAL_GEN_ID **has** SIG_LIST	

Cuando la forma de acotación de los parámetros genéricos formales que se defina para el lenguaje esté dada por acotación mediante subtipado o conformidad, las reglas que se utilizan son las marcadas con S (regla 48S y 49S).

Cuando la forma de acotación de los parámetros genéricos formales que se defina para el lenguaje esté dada por acotación mediante cláusulas *tal que*, las reglas que se utilizan son las marcadas con W (regla 48W, 49W y 50W).

No se da una definición específica para los símbolos que representan terminales indirectos. Estos símbolos son: CLASS_ID, METHOD_ID, VAR_ID, FORMAL_GEN_ID, REAL_CONSTANT, INTEGER_CONSTANT, BOOLEAN_CONSTANT, CHAR_CONSTANT, STRING_CONSTANT.

APÉNDICE B

CARACTERIZACIÓN PROPUESTA APLICADA AL CATÁLOGO DE FOWLER

El caso de estudio seleccionado para aplicar la caracterización es el catálogo básico de [Fowler, 1999] con sus 68 refactorizaciones (sin incluir las denominadas *"big refactoring"*). Una vez validada la aplicabilidad al catálogo se utilizó también en la caracterización del catálogo en genericidad propuesto en el Capítulo 5.

La utilización de este catálogo aporta una gran ventaja puesto que da un punto de vista común, con un vocabulario bien conocido. Es más, aunque las herramientas cambian su denominación, es relativamente simple identificarlas como la misma refactorización al tener similares entradas y los mismos efectos en el estado del código refactorizado.

B.1. Orden en la implementación

La determinación del orden de implementación ante un conjunto amplio de refactorizaciones no es un problema trivial. Sin información adicional no se puede determinar un orden, por lo que debemos aplicar algunos criterios para facilitar dicha labor.

En [Fowler, 1999], se definen relaciones de "uso" entre refactorizaciones, establecidas por el autor desde su experiencia y conocimiento. Las refactorizaciones son agrupadas en base a criterios funcionales, pero en estos grupos no se da un orden de implementación. Tomando el grafo de dependencias resultante de las relaciones de uso extraídas de dicha obra, se puede establecer un ordenamiento topológico, y por lo tanto deducir un orden parcial de las refactorizaciones. Este método es ampliamente utilizado a la hora de desarrollar los módulos

de un sistema en enfoques de abajo a arriba, o *bottom-up*, e igualmente utilizado en las pruebas de sistemas.

Sin embargo nuestra propuesta simplifica el proceso, con similares resultados pero bajo una óptica más objetiva basada en las características de las refactorizaciones, sin requerir una experiencia previa del programador en su aplicación práctica. Para ello se ordenarán de manera decreciente las refactorizaciones tomando como primer criterio, y en este orden, el ámbito, nivel de conocimiento, las entradas (raíz y adicionales) y la acción asociada. En segundo lugar, dentro de cada grupo, se ordena de manera descendente entre los valores de mayor a menor complejidad, de forma similar a la utilización de la cláusula `ORDER BY` en las sentencias SQL.

Inicialmente se aplicará la caracterización al conjunto completo, aunque se presentarán los resultados parciales siguiendo la división funcional ya establecida en [Fowler, 1999], para posteriormente presentar todos los resultados conjuntos en la Tabla B.7.

De manera general y por motivos de simplicidad, se eliminan en todos los grafos aquellas refactorizaciones que no tienen explícitamente ninguna relación de uso con otras refactorizaciones y forman nodos aislados.

En el grupo Composing Methods, tenemos 9 refactorizaciones con su grafo de dependencias (ver Fig. B.1). Utilizando la caracterización se obtiene la ordenación parcial de las refactorizaciones de mayor a menor complejidad (ver Tabla B.1), donde el orden parcial obtenido coincide con el del grafo salvo en la ordenación parcial entre **Replace Temp with Query** y **Split Temporary Variable**. Por otro lado se obtiene información no disponible en el grafo, como la ordenación de las refactorizaciones **Introduce Explaining Variable** y **Substitute Algorithm**, de las cuales no podíamos colegir ninguna conclusión en [Fowler, 1999].

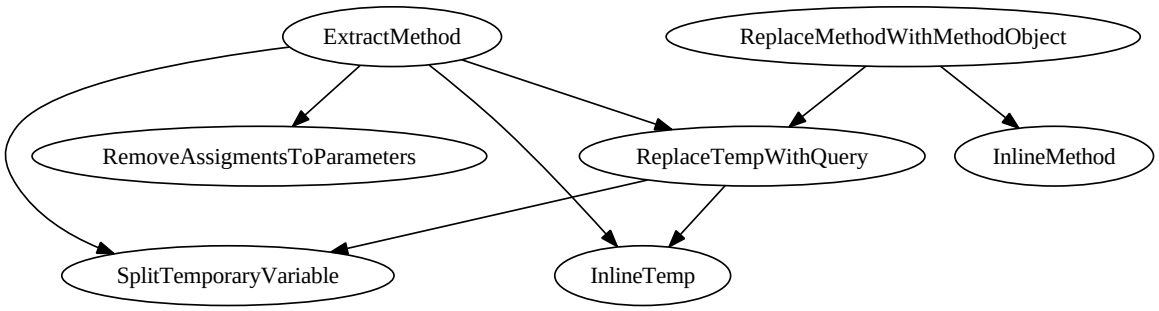

Figura B.1: Refactorizaciones relacionadas en el grafo Composing Methods (7 de 9)

En el grupo Moving Features Between Objects, tenemos 8 refactorizaciones con su correspondiente grafo de dependencias (ver Fig. B.2). Utilizando la caracterización se obtiene el resultado mostrado en la Tabla B.2. El orden obtenido coincide con uno de los ordenes parciales que podría ser deducido de una ordenación topológica del grafo. Con la

Tabla B.1: Orden parcial de mayor a menor complejidad del grupo Composing Methods

	Refactoring	Scope	DLI	Root input	Additional Inputs	Action
1	Replace Method with Method Object	IC	Basic	Method	1 Name	Replace
2	Substitute Algorithm	I	Basic	Method	* Instructions	Replace
3	Inline Method	I	Basic	Method	No additional inputs	Replace
4	Extract Method	I	Basic	Code Fragment	1 Name	Replace
5	Introduce Explaining Variable	I	Basic	Expression	* Names	Replace
6	Split Temporary Variable	I	Basic	Expression	* Names	Replace
7	Remove Assignment to Parameters	I	Basic	Formal Argument	1 Name	Replace
8	Replace Temp with Query	I	Basic	Local Declaration	1 Name	Replace
9	Inline Temp	I	Basic	Local Declaration	No additional inputs	Replace

caracterización se consigue además ordenar las refactorizaciones **Remove Middle Man** e **Introduce Foreign Method**, de las que no se disponía de información inicialmente.

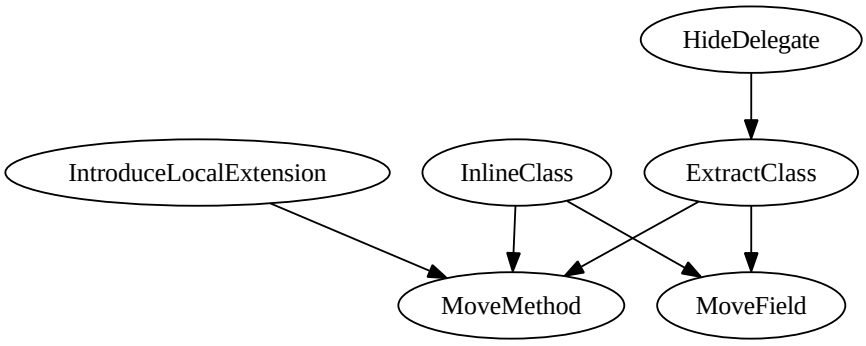

Figura B.2: Refactorizaciones relacionadas en el grafo Moving Features Between Objects (6 de 8)

Tabla B.2: Orden parcial de mayor a menor complejidad del grupo Moving Features Between Objects

	Refactoring	Scope	DLI	Root input	Additional Inputs	Action
1	Hide Delegate	ICH	Design Pattern	Class	* Classes	Move
2	Remove Middle Man	ICH	Design Pattern	Class	* Classes	Remove
3	Introduce Local Extension	ICH	Basic	Class	1 Class * Methods	Add
4	Inline Class	ICH	Basic	Class	1 Class	Move
5	Extract Class	ICH	Basic	Class	* Attributes	Move
6	Move Method	ICH	Basic	Method	1 Class	Move
7	Move Field	ICH	Basic	Attribute	1 Class	Move
8	Introduce Foreign Method	IC	Basic	Class	* Instructions	Add

En el grupo Organizing Data, tenemos 16 refactorizaciones con un grafo de dependencias simplificado como se muestra en la Fig. B.3. Debido a la ausencia de dependencias entre las refactorizaciones del grupo, se ha eliminado un gran grupo de nodos del grafo que no aportaban información respecto a una posible ordenación.

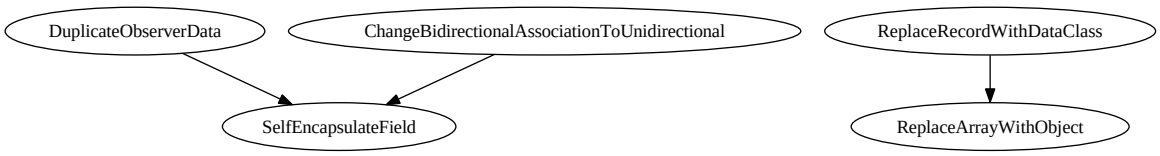

Figura B.3: Refactorizaciones relacionadas en el grafo Organizing Data (5 de 16)

Utilizando la caracterización se obtiene la ordenación mostrada en la Tabla B.3. De nuevo, el orden obtenido coincide con una ordenación parcial de los nodos del grafo, pero además este ejemplo permite mostrar claramente la ventaja de utilizar una caracterización como la propuesta. Con un número medio de refactorizaciones (16) y con pocas interdependencias entre las mismas (como se desprende del grafo en la Fig. B.3), la caracterización permite una ordenación (aunque con orden parcial) de todos los elementos del grupo.

En el grupo Simplifying Conditional Expression, tenemos 8 refactorizaciones. No se representa el grafo dado que en este grupo no existen dependencias de uso entre las refactorizaciones, por lo que estamos ante una situación de ausencia de información a la

Tabla B.3: Orden parcial de mayor a menor complejidad del grupo Organizing Data

	Refactoring	Scope	DLI	Root input	Additional Inputs	Action
1	Duplicate Observer Data	ICH	Design Pattern	Class	* Properties	Move
2	Replace Type Code With State/Strategy	ICH	Design Pattern	Class	No additional inputs	Move
3	Replace Subclass With Fields	ICH	Design Pattern	Class	No additional inputs	Move
4	Replace Type Code with Subclasses	ICH	Basic	Class	No additional inputs	Move
5	Encapsulate Field	ICH	Basic	Attribute	No additional inputs	Replace
6	Change Value to Reference	IC	Design Pattern	Class	1 Class	Replace
7	Change Reference to Value	IC	Design Pattern	Class	1 Class	Replace
8	Change Unidirectional Association to Bidirectional	IC	Basic	Class	1 Class	Replace
9	Change Bidirectional Association to Unidirectional	IC	Basic	Class	1 Class	Replace
10	Replace Type Code with Class	IC	Basic	Class	* Attributes	Move
11	Replace Record with Data Class	IC	Basic	Class	* Attributes	Move
12	Replace Array with Object	IC	Basic	Entity	* Names	Replace
13	Replace Data Value with Object	IC	Basic	Attribute	No additional inputs	Move
14	Encapsulate Collection	I	Basic	Method	No additional inputs	Replace
15	Replace Magic Number with Symbolic Constant	I	Basic	Expression	1 Name	Replace
16	Self Encapsulate Field	I	Basic	Attribute	No additional inputs	Replace

hora de afrontar una ordenación de las refactorizaciones. La existencia de un gran número de posibles ordenaciones parciales no facilita la tarea de planificar una implementación ordenada.

Utilizando la caracterización se obtiene el orden mostrado en la Tabla B.4. Obviamente los resultados respetan un orden parcial del grafo, pero además ordenan en base a criterios más objetivos las refactorizaciones dentro del grupo.

Tabla B.4: Orden parcial de mayor a menor complejidad del grupo Simplifying Conditional Expressions

	Refactoring	Scope	DLI	Root input	Additional Inputs	Action
1	Introduce Null Object	ICH	Basic	Class	No additional inputs	Move
2	Replace Conditional with Polymorphism	ICH	Basic	Code Fragment	* Classes	Move
3	Introduce Assertion	I	Advanced	Expression	No additional inputs	Add
4	Consolidate Duplicate Conditional Fragments	I	Basic	Code Fragment	No additional inputs	Move
5	Consolidate Conditional Expression	I	Basic	Code Fragment	No additional inputs	Replace
6	Replace Nested Conditional with Guard Clauses	I	Basic	Code Fragment	No additional inputs	Replace
7	Decompose Conditional	I	Basic	Expression	No additional inputs	Move
8	Remove Control Flag	I	Basic	Local Declaration	No additional inputs	Move

En el grupo Making Method Calls Simpler, tenemos 15 refactorizaciones con un grafo de dependencias entre ellas (ver Fig. B.4). De nuevo estamos ante la situación de que muchas refactorizaciones no tienen dependencias con otras del grupo, dando un grafo con excesivo número de nodos aislados (eliminados de la figura).

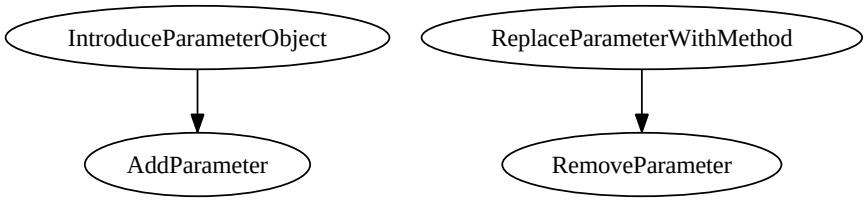

Figura B.4: Refactorizaciones relacionadas en el grafo Making Method Calls Simpler (4 de 15)

Utilizando la caracterización se obtiene la ordenación mostrada en la Tabla B.5. De nuevo se puede observar que se resuelve la posición de un gran número de refatorizaciones (11) de las que inicialmente en el grafo no podíamos deducir nada.

Tabla B.5: Orden parcial de mayor a menor complejidad del grupo Making Method Calls Simpler

	Refactoring	Scope	DLI	Root input	Additional Inputs	Action
1	Replace Constructor with Factory Method	ICH	Design Pattern	Method	No additional inputs	Replace
2	Replace Error Code with Exception	ICH	Advanced	Method	1 Class	Replace
3	Replace Exception with Test	ICH	Advanced	Code Fragment	No additional inputs	Replace
4	Parameterize Method	ICH	Basic	Method	* Methods	Move
5	Introduce Parameter Object	ICH	Basic	Method	* Methods	Move
6	Rename Method	ICH	Basic	Method	1 Name	Rename
7	Replace Parameter with Method	ICH	Basic	Method	No additional inputs	Move
8	Encapsulate Downcast	ICH	Basic	Method	No additional inputs	Move
9	Separate Query from Modifier	ICH	Basic	Method	No additional inputs	Move

(continúa en la página siguiente...)

Tabla B.5 – (viene de la página anterior)

10	Preserve Whole Object	ICH	Basic	Method	No additional inputs	Replace
11	Remove Setting Method	ICH	Basic	Attribute	No additional inputs	Remove
12	Replace Parameter with Explicit Methods	ICH	Basic	Formal Argument	No additional inputs	Move
13	Remove Parameter	ICH	Basic	Formal Argument	No additional inputs	Remove
14	Add Parameter	ICH	Basic	Formal Argument	No additional inputs	Add
15	Hide Method	IC	Basic	Method	No additional inputs	Replace

En el grupo Dealing with Generalization, tenemos 12 refactorizaciones con con el grafo de dependencias mostrado en la Fig. B.5.

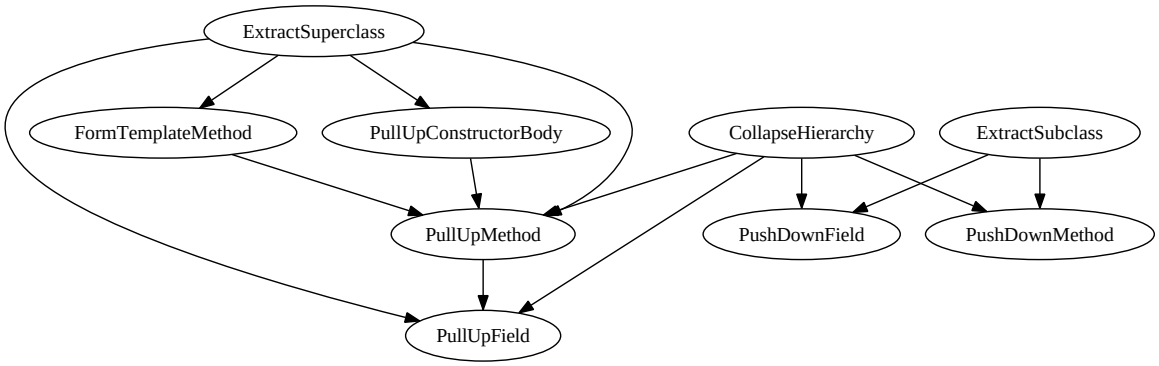

Figura B.5: Refactorizaciones relacionadas en el grafo Dealing with Generalization (9 de 12)

Utilizando la caracterización se obtiene la ordenación mostrada en la Tabla B.6. De nuevo, se obtiene un orden parcial coincidente con uno de los existentes en el grafo, pero adicionalmente se sitúan refactorizaciones de las que no se disponía información como **Extract**

Interface, **Replace Inheritance With Delegation** y **Replace Delegation With Inheritance**.

Tabla B.6: Orden parcial de mayor a menor complejidad del grupo Dealing with Generalization

	Refactoring	Scope	DLI	Root input	Additional Inputs	Action
1	Extract Subclass	ICH	Design Pattern	Class	* Classes	Move
2	Extract Superclass	ICH	Design Pattern	Class	* Classes	Move
3	Replace Inheritance with Delegation	ICH	Design Pattern	Class	1 Class	Replace
4	Replace Delegation with Inheritance	ICH	Design Pattern	Class	1 Class	Replace
5	Form Template Method	ICH	Design Pattern	Method	* Methods	Move
6	Collapse Hierarchy	ICH	Basic	Class	* Classes	Move
7	Extract Interface	ICH	Basic	Class	* Methods	Move
8	Push Down Method	ICH	Basic	Method	No additional inputs	Move
9	Push Down Field	ICH	Basic	Attribute	No additional inputs	Move
10	Pull Up Constructor Body	IH	Basic	Method	* Methods	Move
11	Pull Up Method	IH	Basic	Method	No additional inputs	Move
12	Pull Up Field	IH	Basic	Attribute	No additional inputs	Move

Los resultados obtenidos en estas ordenaciones permiten su comparación con la información disponible en [Fowler, 1999], utilizando sus grupos en base a su propia caracterización basada en la similitud de funcionalidad. Sin embargo, la caracterización y el método pro-

puesto permite su aplicación de una manera bastante simple al conjunto completo de 68 refactorizaciones. El resultado obtenido se muestra en la Tabla B.7 donde se respetan los órdenes parciales ya comentados dentro de cada grupo, pero se aporta una ordenación parcial completa sobre todas las refactorizaciones del catálogo de mayor a menor complejidad, siendo uno de los principales objetivos a cumplir.

Tabla B.7: Orden parcial de mayor a menor complejidad del catálogo en [Fowler, 1999]

	Refactoring	Scope	DLI	Root input	Additional Inputs	Action
1	Extract Subclass	ICH	Design Pattern	Class	* Classes	Move
2	Extract Superclass	ICH	Design Pattern	Class	* Classes	Move
3	Hide Delegate	ICH	Design Pattern	Class	* Classes	Move
4	Remove Middle Man	ICH	Design Pattern	Class	* Classes	Remove
5	Replace Inheritance with Delegation	ICH	Design Pattern	Class	1 Class	Replace
6	Replace Delegation with Inheritance	ICH	Design Pattern	Class	1 Class	Replace
7	Duplicate Observer Data	ICH	Design Pattern	Class	* Properties	Move
8	Replace Type Code With State/Strategy	ICH	Design Pattern	Class	No additional inputs	Move
9	Replace Subclass With Fields	ICH	Design Pattern	Class	No additional inputs	Move
10	Form Template Method	ICH	Design Pattern	Method	* Methods	Move
11	Replace Constructor with Factory Method	ICH	Design Pattern	Method	No additional inputs	Replace
12	Replace Error Code with Exception	ICH	Advanced	Method	1 Class	Replace

(continúa en la página siguiente...)

Tabla B.7 – (viene de la página anterior)

13	Replace Exception with Test	ICH	Advanced	Code Fragment	No additional inputs	Replace
14	Introduce Local Extension	ICH	Basic	Class	1 Class * Methods	Add
15	Collapse Hierarchy	ICH	Basic	Class	* Classes	Move
16	Inline Class	ICH	Basic	Class	1 Class	Move
17	Extract Interface	ICH	Basic	Class	* Methods	Move
18	Extract Class	ICH	Basic	Class	* Attributes	Move
19	Introduce Null Object	ICH	Basic	Class	No additional inputs	Move
20	Replace Type Code with Subclasses	ICH	Basic	Class	No additional inputs	Move
21	Move Method	ICH	Basic	Method	1 Class	Move
22	Parameterize Method	ICH	Basic	Method	* Methods	Move
23	Introduce Parameter Object	ICH	Basic	Method	* Methods	Move
24	Rename Method	ICH	Basic	Method	1 Name	Rename
25	Push Down Method	ICH	Basic	Method	No additional inputs	Move
26	Replace Parameter with Method	ICH	Basic	Method	No additional inputs	Move
27	Encapsulate Downcast	ICH	Basic	Method	No additional inputs	Move
28	Separate Query from Modifier	ICH	Basic	Method	No additional inputs	Move
29	Preserve Whole Object	ICH	Basic	Method	No additional inputs	Replace
30	Replace Conditional with Polymorphism	ICH	Basic	Code Fragment	* Classes	Move

(continúa en la página siguiente...)

Tabla B.7 – (viene de la página anterior)

31	Move Field	ICH	Basic	Attribute	1 Class	Move
32	Push Down Field	ICH	Basic	Attribute	No additional inputs	Move
33	Encapsulate Field	ICH	Basic	Attribute	No additional inputs	Replace
34	Remove Setting Method	ICH	Basic	Attribute	No additional inputs	Remove
35	Replace Parameter with Explicit Methods	ICH	Basic	Formal Argument	No additional inputs	Move
36	Remove Parameter	ICH	Basic	Formal Argument	No additional inputs	Remove
37	Add Parameter	ICH	Basic	Formal Argument	No additional inputs	Add
38	Pull Up Constructor Body	IH	Basic	Method	* Methods	Move
39	Pull Up Method	IH	Basic	Method	No additional inputs	Move
40	Pull Up Field	IH	Basic	Attribute	No additional inputs	Move
41	Change Value to Reference	IC	Design Pattern	Class	1 Class	Replace
42	Change Reference to Value	IC	Design Pattern	Class	1 Class	Replace
43	Change Unidirectional Association to Bidirectional	IC	Basic	Class	1 Class	Replace
44	Change Bidirectional Association to Unidirectional	IC	Basic	Class	1 Class	Replace
45	Replace Type Code with Class	IC	Basic	Class	* Attributes	Move

(continúa en la página siguiente...)

Tabla B.7 – (viene de la página anterior)

46	Replace Record with Data Class	IC	Basic	Class	* Attributes	Move
47	Introduce Foreign Method	IC	Basic	Class	* Instructions	Add
48	Replace Method with Method Object	IC	Basic	Method	1 Name	Replace
49	Hide Method	IC	Basic	Method	No additional inputs	Replace
50	Replace Array with Object	IC	Basic	Entity	* Names	Replace
51	Replace Data Value with Object	IC	Basic	Attribute	No additional inputs	Move
52	Introduce Assertion	I	Advanced	Expression	No additional inputs	Add
53	Substitute Algorithm	I	Basic	Method	* Instructions	Replace
54	Encapsulate Collection	I	Basic	Method	No additional inputs	Replace
55	Inline Method	I	Basic	Method	No additional inputs	Replace
56	Extract Method	I	Basic	Code Fragment	1 Name	Replace
57	Consolidate Duplicate Conditional Fragments	I	Basic	Code Fragment	No additional inputs	Move
58	Consolidate Conditional Expression	I	Basic	Code Fragment	No additional inputs	Replace
59	Replace Nested Conditional with Guard Clauses	I	Basic	Code Fragment	No additional inputs	Replace
60	Introduce Explaining Variable	I	Basic	Expression	* Names	Replace
61	Split Temporary Variable	I	Basic	Expression	* Names	Replace

(continúa en la página siguiente...)

Tabla B.7 – (viene de la página anterior)

62	Replace Magic Number with Symbolic Constant	I	Basic	Expression	1 Name	Replace
63	Decompose Conditional	I	Basic	Expression	No additional inputs	Move
64	Self Encapsulate Field	I	Basic	Attribute	No additional inputs	Replace
65	Remove Assignment to Parameters	I	Basic	Formal Argument	1 Name	Replace
66	Replace Temp with Query	I	Basic	Local Declaration	1 Name	Replace
67	Remove Control Flag	I	Basic	Local Declaration	No additional inputs	Move
68	Inline Temp	I	Basic	Local Declaration	No additional inputs	Replace

APÉNDICE C

ESPECIFICACIÓN DE FUNCIONES EN ALF

Tras haber establecido en el Capítulo 5, y en particular en la Sec. 5.1, las propiedades que adecúan el uso del lenguaje ALF a la definición de refactorizaciones, se define a continuación el conjunto de funciones básicas siguiendo dicha notación. La corrección de la definición ha sido validada sobre los elementos del metamodelo MOON definidos en el Capítulo 3.

La definición se restringe sólo a los elementos contenidos en el metamodelo MOON, con el objetivo de obtener una cierta independencia del lenguaje.

C.1. `DependantDownFormalPar`

Descripción: dado un parámetro formal de una clase, devuelve todos los parámetros formales dependientes hacia abajo en clases **descendientes** y **clientes**.

A continuación se detalla la especificación de la actividad `DependantDownFormalPar` en ALF, que depende de la actividad denominada `CollectDependantDownFormalPar`. El objetivo de dicha función es recopilar los parámetros formales dependientes en descendientes y clientes, repitiendo recursivamente la operación hasta completar la lista de parámetros formales.

Código C.1.1: Función `DependantDownFormalPar`

```
namespace Moon::Function;

/*
 * Gets the dependant down formal parameters.
 */
```

```
activity DependantDownFormalPar(in formalPar : FormalPar , inout visitedClasses: List<
    ClassDef>, inout visitedFormalParameters : List<FormalPar>) : FormalPar[0..*]
    sequence {

        if (!visitedClasses -> contains(formalPar.getClassDef())) {
                visitedFormalParameters -> including(formalpar);

                // Clients
                List<ClassType> directTypeClients = formalPar.getClassDef().
                    getDirectTypeClient();
                CollectDependantDownFormalPar(formalPar, directTypeClients,
                    visitedClasses,  visitedFormalParameters);

                // Descendants
                List<ClassDef> directDescendants = formalPar.getClassDef().
                    getDirectDescendants();
                List<ClassType> directTypeDescendants = new List<ClassType>();
                for (ClassDef cd : directDescendants) {
                        directTypeDescendants.add(cd.getClassTypeWithParameters());
                }
                CollectDependantDownFormalPar(formalPar, directTypeDescendants,
                    visitedClasses,      visitedFormalParameters);

                // Add the visited class
                visitedClasses -> including(formalPar.getClassDef());
        } // if
}
```

La actividad `CollectDependantDownFormalPar`, repite el proceso de búsqueda para los tipos dependientes (descendientes o clientes), buscando la correspondiente sustitución del parámetro formal por posición. En caso de que la sustitución sea un parámetro formal en la clase descendiente o cliente, se lanza recursivamente la búsqueda de parámetros formales dependientes hacia abajo:

Código C.1.2: Función `CollectDependantDownFormalPar`

```
namespace Moon::Function;

/*
 * Collects the dependant down formal parameters.
 */
activity CollectDependantDownFormalPar(in formalPar : FormalPar , in directTypes : List
    <ClassType>, inout visitedClasses : List<ClassDef>, inout visitedFormalParameters :
     List<FormalPar>) {
        for (ClassType dependantType : directTypes) {
           if (dependantType.isParametricType() && !dependantType.isComplete()){
                // Get the equivalent formal parameter by position
                Type type = dependantType.getRealParameters() -> at(formalPar.getIndex
                    ());
               if (!type.isComplete() && type instanceof FormalPar) {
                // Indirect recursive call
                        DependantDownFormalPar((FormalPar) type, visitedClasses,
                            visitedFormalParameters);
                  } //if
           } // if
        } // for
}
```

C.2. RealSubstitutionsOfFormalPar

Descripción: dado un parámetro formal de una clase, devuelve todas las sustituciones por tipos definidos por clases (`ClassType`) a dicho parámetro formal.

Código C.2.3: Función `RealSubstitutionsOfFormalPar`

```
namespace Moon::Function;

/*
 * Gets the real substitutions of the formal parameter.
 */
activity RealSubstitutionsOfFormalPar(in formalPar : FormalPar) : ClassType[0..*]
    sequence {

        List<ClassType> result = new List<ClassType>();

        /* if the type is parametric and has as base class the same class
                that declares the formal parameter, find the substitutions */
        List<Type> types = Type -> select t (t.isParametric() && t.getClassDef() ==
            formalPar.getClassDef());

        for (Type type : types) {
                Type typeAux = (Type) (((ClassType) type).getRealParameters() -> at(
                    formalPar.getIndex()));
                if (typeAux instanceof ClassType) {
                        result -> including(typeAux);
                } // if
        } // for
        return result;
}
```

C.3. CollectSignaturesToAdd

Descripción: dado un parámetro formal de una clase y, a partir del conjunto de tipos definidos por clase que se utilizan como sustitución del parámetro formal obtenidos con la función `RealSubstitutionsOfFormalPar` y de los parámetros formales dependientes hacia abajo, calculados con la función `DependantDownFormalPar`, calcula el conjunto de métodos que tienen todos estos tipos en común a través de la función `CollectMethodIntersection`.

Código C.3.4: Función `CollectSignaturesToAdd`

```
namespace Moon::Function;

/*
 * Gets the common methods with the same signature in a set of class types.
 */
activity CollectSignaturesToAdd(in formalPar : FormalPar) : List<Method>  {

        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        List<FormalPar> formalParameters = new List<FormalPar>();
```

```
        formalParameters.addAll(DependantDownFormalPar(formalPar, visitedClasses,
            visitedFormalParameters));

        List<ClassType> classTypes = new List<ClassType>();
        for (FormalPar fp : formalParameters) {
                classTypes.addAll(RealSubstitutionsOfFormalPar(fp));
        }
        return CollectMethodIntersection(classTypes);
}
```

C.4. CollectMethodIntersection

Descripción: dado un conjunto de tipos definidos por clase, calcula la intersección del conjunto de métodos que tienen igual signatura y pertenecen a todas las clases determinantes de dichos tipos.

Código C.4.5: Función `CollectMethodIntersection`

```
namespace Moon::Function;


/*
 * Gets the common methods with the same signature in a set of class types.
 */
activity CollectMethodIntersection(in classes : List<ClassType>) : List<Method>  {
        List<Method> methods = new List<Method>();
        List<ClassDef> othersClassDef = new List<ClassDef>();
        // Collect the classes of the types
        othersClassDef.addAll(classes.toSequence().getClassDef());

        for (ClassDef classDef : othersClassDef) {
                List<Property> otherProperties = classDef.getBasicShortFlattened();
                // for each method in the basic short flat form (without properties
                    from universal object)
                for (Property prop : otherProperties) {
                        if (prop instanceof Method) { // if it is a method
                                if (! methods.toSequence() -> exists m (m.
                                    hasSameSignature((Method) prop))
                                    // not exists a method in the list with same
                                        signature
                                    && othersClassDef.toSequence() -> excluding( ((
                                        Method) prop).getClassDef()) ->
                                                forAll cls (cls.getMethods().
                                                    toSequence()
                                                        -> exists m (m.
                                                            hasSameSignature((
                                                            Method)prop))))
                                    // and all other classes have a method with
                                        same signature
                                {
                                    methods.add((Method) prop); // add the method
                                }
                        } // if it is a method
                } // for all properties
        } // for all classes
        return methods;
```

```
}
```

C.5. DependantDescendantsProvidersBoundW

Descripción: dado un parámetro formal con acotación por cláusulas *tal que* de una clase, devuelve todos los parámetros formales dependientes hacia abajo en clases descendientes, así como los parámetros formales dependientes en proveedores de dichas clases, siempre y cuando contengan el método en su acotación.

Código C.5.6: Función `DependantDescendantsProvidersBoundW`

```
namespace Moon::Function;

public import Moon::ClassType;

/*
 * Gets the dependant formal parameters in descendants and providers with a method.
 */
activity DependantDescendantsProvidersBoundW(in boundW : BoundW, inout visitedClasses:
    List<ClassDef>,
                        inout visitedBoundWParameters : List<BoundW>, in method :
                            Method) : List<BoundW> {

        if (! (visitedClasses.toSequence() -> includes(boundW.getClassDef()))) {

                visitedBoundWParameters.add(boundW);

                // Providers
                List<ClassType> directTypeProviders = boundW.getClassDef().
                    getDirectTypeProviders();

                CollectDependantProvidersBoundW(boundW, directTypeProviders,
                    visitedClasses,
                        visitedBoundWParameters, method);

                // Descendants
                List<ClassDef> directDescendants = new List<ClassDef>();
                directDescendants.addAll(boundW.getClassDef().getDirectDescendants());

                List<ClassType> directTypeDescendants = new List<ClassType>();
                for (ClassDef cd : directDescendants) {
                        directTypeDescendants.add(cd.getClassTypeWithParameters());
                }

                CollectDependantDescendantsBoundW(boundW, directTypeDescendants,
                    visitedClasses,
                        visitedBoundWParameters, method);

                // Add the visited class
                visitedClasses -> including(boundW.getClassDef());
        } // if
        return visitedBoundWParameters;
}
```

Depende de dos funciones auxiliares: `CollectDependantProvidersBoundW` y `CollectDependantDescendantsBoundW`.

La actividad `CollectDependantProvidersBoundW`, repite el proceso de búsqueda para los tipos dependientes en proveedores, siempre y cuando contengan el método indicado en la acotación.

Código C.5.7: Función `CollectDependantProvidersBoundW`

```
namespace Moon::Function;

/*
 * Collects providers that include the method in the initial where clause as bound.
 */
activity CollectDependantProvidersBoundW(in boundW : BoundW , in directProviders : List
    <ClassType>,
                inout visitedClasses : List<ClassDef>, inout visitedBoundWParameters :
                    List<BoundW>, in method : Method) {

        for (ClassType providerType : directProviders) {
                if (providerType.isParametric() && !providerType.isComplete()){
                        Type type = providerType.getRealParameters() -> at(boundW.
                            getIndex());
                        if (! type.isComplete() && type instanceof BoundW &&
                                        (((BoundW) type).getProperties().toSequence()
                                            -> includes(method))) {
                                // Recursive calling
                                DependantDescendantsProvidersBoundW((BoundW) type,
                                    visitedClasses,
                                            visitedBoundWParameters, method);
                        } //if
                } // if
        } // for
}
```

La actividad `CollectDependantDescendantsBoundW`, repite el proceso de búsqueda para los tipos dependientes en los descendientes.

Código C.5.8: Función `CollectDependantDescendantsBoundW`

```
namespace Moon::Function;

/*
 * Visits the descendants with an equivalent boundW parameter.
 */
activity CollectDependantDescendantsBoundW(in boundW : BoundW , in descendantTypes :
    List<ClassType>,
                inout visitedClasses : List<ClassDef>, inout visitedBoundWParameters :
                    List<BoundW>, in method : Method) {

        for (ClassType dependantType : descendantTypes) {
                if (dependantType.isParametric() && !dependantType.isComplete()){
                        Type type = dependantType.getRealParameters() -> at(boundW.
                            getIndex());
                        if (! type.isComplete() && type instanceof BoundW) {
                                // Recursive calling
                                DependantDescendantsProvidersBoundW((BoundW) type,
                                    visitedClasses,
                                            visitedBoundWParameters, method);
                        } //if
                } // if
        } // for
}
```

C.6. CollectEntitiesWithType

Descripción: dado un parámetro formal de una clase, devuelve el conjunto de entidades de la clase cuyo tipo es igual al parámetro genérico formal.

Código C.6.9: Función `CollectEntitiesWithType`

```
namespace Moon::Function;

/*
 * Collects the entities with type formal parameter in the class that contains the
     formal parameter.
 */
activity CollectEntitiesWithType(in formalPar : FormalPar) :  List<Entity> {
        List<Entity> entities = formalPar.getClassDef().getEntities();
        List<Entity> entitiesResult = new List<Entity>();
        for (Entity entity : entities) {
                if (entity.getType() == formalPar){
                        entitiesResult.add(entity);
                } // if
        } // for
        return entitiesResult;
}
```

C.7. CollectMethodsUsedByEntity

Descripción: dada una entidad, devuelve el conjunto de métodos invocados a través de dicha entidad.

Código C.7.10: Función `CollectMethodsUsedByEntity`

```
namespace Moon::Function;

/*
 * Collects the methods used by an entity.
 */
activity CollectMethodsUsedByEntity(in entity : Entity) :  List<Method> {
        List<Method> methods = new List<Method>();
        List<CallExpr> expressions = entity.getClassDef().getExpressions();
        List<CallInstr> instructions = entity.getClassDef().getInstructions();


        // Methods used in expression with current entity
        for (CallExpr callExpr : expressions) {
                if (callExpr.getTarget() == entity && callExpr.getRightSide()
                    instanceof CallExpr
                        && ((CallExpr) (callExpr.getRightSide())).getTarget()
                            instanceof Result) {
                         Result result = (Result) ((CallExpr) (callExpr.getRightSide())
                             ).getTarget();
                         methods.add(result.getFunctionDec());
                }
        }

        // Methods used in expression with current instruction
        for (CallInstr callInstr : instructions)  {
```

```
                if (callInstr.getLeftSide().getTarget() == entity) {
                        methods.add(callInstr.getRoutineDec());
                }
        }
        return methods;
}
```

C.8. DependantUpFormalPar

Descripción: dado un parámetro formal de una clase, devuelve todos los parámetros formales dependientes hacia arriba en clases **ancestros** y **proveedores**.

El objetivo de dicha función es recursivamente recopilar los parámetros formales dependientes en ancestros y proveedores, transitivamente repitiendo la operación hasta completar la lista de parámetros formales.

Código C.8.11: Función DependantUpFormalPar

```
namespace Moon::Function;

/*
 * Gets the dependant formal parameters in ancestors and providers.
 */
activity DependantUpFormalPar(in formalPar : FormalPar , inout visitedClasses: List<
    ClassDef>,
                        inout visitedFormalParameters : List<FormalPar>) : List<
                            FormalPar> {

        if (! (visitedClasses.toSequence() -> includes(formalPar.getClassDef()))) {

                visitedFormalParameters.add(formalPar);

                // Providers
                List<ClassType> directTypeProviders = formalPar.getClassDef().
                    getDirectTypeProviders();

                CollectDependantUpFormalPar(formalPar, directTypeProviders,
                    visitedClasses,
                        visitedFormalParameters);

                // Ancestors
                List<ClassType> directTypeAncestors = formalPar.getClassDef().
                    getDirectTypeAncestors();

                CollectDependantUpFormalPar(formalPar, directTypeAncestors,
                    visitedClasses,
                        visitedFormalParameters);

                // Add the visited class
                visitedClasses -> including(formalPar.getClassDef());
        } // if
        return visitedFormalParameters;
}
```

La actividad CollectDependantUpFormalPar, repite el proceso de búsqueda para los tipos dependientes, lanzando de nuevo la búsqueda de parámetros formales dependientes:

Código C.8.12: Función `CollectDependantUpFormalPar`

```
namespace Moon::Function;

/*
 * Collects the dependant up formal parameters.
 */
activity CollectDependantUpFormalPar(in formalPar : FormalPar , in directTypes : List<
    ClassType>,
                inout visitedClasses : List<ClassDef>, inout visitedFormalParameters :
                    List<FormalPar>) {

        for (ClassType dependantType : directTypes) {
                if (dependantType.isParametric() && !dependantType.isComplete()){
                        Type type = dependantType.getRealParameters() -> at(formalPar.
                            getIndex());
                        if (! type.isComplete() && type instanceof FormalPar) {
                                // Recursive calling
                                DependantUpFormalPar(FormalPar) type, visitedClasses,
                                    visitedFormalParameters);
                        } //if
                } // if
        } // for
}
```

C.9. CollectDownRealSubstitutions

Descripción: dado un parámetro formal devuelve todas las sustituciones por tipos definidos por clases en parámetros formales dependientes hacia abajo.

Código C.9.13: Función `CollectDownRealSubstitutions`

```
namespace Moon::Function;

public import Moon::Property;
public import Moon::Method;
public import Moon::ClassDef;
public import Moon::ClassType;
public import Moon::FormalPar;

/*
 * Gets the class types substitutions of formal parameters in down direction.
 */
activity CollectDownRealSubstitutions(in formalPar : FormalPar) : List<ClassType>  {

        List<ClassDef> visitedClasses = new List<ClassDef>();
        List<FormalPar> visitedFormalParameters = new List<FormalPar>();

        List<FormalPar> formalParameters = new List<FormalPar>();
        formalParameters.addAll(DependantDownFormalPar(formalPar, visitedClasses,
            visitedFormalParameters));

        List<ClassType> classTypes = new List<ClassType>();
        for (FormalPar fp : formalParameters) {
                classTypes.addAll(RealSubstitutionsOfFormalPar(fp));
        }
        return classTypes;
}
```

C.10. CollectPropertiesUsedByEntity

Descripción: dada una entidad, devuelve el conjunto de las propiedades invocadas por dicha entidad en el conjunto de instrucciones.

Código C.10.14: Función `CollectPropertiesUsedByEntity`

```
namespace Moon::Function;

/*
 * Collects the properties used by an entity.
 */
activity CollectPropertiesUsedByEntity(in entity : Entity) :  List<Property> {
        List<Property> properties = new List<Property>();
        List<CallExpr> expressions = entity.getClassDef().getExpressions();
        List<CallInstr> instructions = entity.getClassDef().getInstructions();
        Boolean condition = false;
        // Properties (functions and attributes) used in expression with current entity
        for (CallExpr callExpr : expressions) {
                if (callExpr.getTarget() == entity && callExpr.getRightSide()
                    instanceof CallExpr) {
                        if( ((CallExpr) (callExpr.getRightSide())).getTarget()
                            instanceof Result) {
                                // Function
                                Result result = (Result) ((CallExpr) (callExpr.
                                    getRightSide())).getTarget();
                                properties.add((Property) result.getFunctionDec());
                        } else if (((CallExpr) (callExpr.getRightSide())).getTarget()
                            instanceof Attribute){
                                // Attribute
                                properties.add((Property) ((CallExpr) (callExpr.
                                    getRightSide())).getTarget());
                        }
                }
        }


        // Properties (routines) used in expression with current instruction
        for (CallInstr callInstr : instructions)  {
                if (callInstr.getLeftSide().getTarget() == entity) {
                        properties.add(callInstr.getRoutineDec());
                }
        }
        return properties;
}
```

APÉNDICE D

ESPECIFICACIÓN DE LA REFACTORIZACIÓN PARAMETRIZAR EN ALF

Dada la complejidad de la refactorización **Parametrizar** definida en el Capítulo 5 (ver Apartado 5.3.1), se detalla a continuación el conjunto de funciones básicas y acciones utilizadas en la definición de dicha refactorización.

D.1. `ParameterizeUniverse`

Descripción: dada una clase devuelve el conjunto de clases condicionantes (ancestros y proveedores), al que se añaden todas las clases descendientes de la clase objetivo[1]. Recursivamente estarán todas las clases condicionantes de las clases incluidas y todas las clases descendientes de las clases incluidas, con excepción de la clase universal (`Object`) (puesto que todas las clases son descendientes de ésta, la inclusión de sus descendientes sería el grafo completo de clases) [Crespo, 2000, p. 153].

Determina el conjunto de clase objeto de estudio a la hora de buscar entidades genérico dependientes directas o indirectas respecto a la entidad de entrada de la refactorización.

Código D.1.1: Función `ParameterizeUniverse`

```
namespace Moon::Function;

activity ParameterizeUniverse(in classDef : ClassDef) : List<ClassDef> {
```

[1]La inclusión de las clases descendientes se realiza por la asunción de que todo cambio será propagado a los descendientes.

```
        List<ClassDef> universe = new List<ClassDef>();
        ClassDef universal = Model.getModel().getUniversalObject();
        // get ancestors and providers
        List<ClassDef> conditioners = classDef.getConditioner();
        conditioners.add(classDef); // Add current class
        conditioners.remove(universal); // Minus Object

        Boolean changes = true;
        while(changes) { // fix point
                Integer oldSize = conditioners.size();
                List<ClassDef> descendants = new List<ClassDef>();
                for (ClassDef cd : conditioners){ // Ci in Vi
                        descendants -> union(cd.getDescendants());
                }
                List<ClassDef> conditionants = new List<ClassDef>();
                for (ClassDef cd : descendants){ // Ci in Descencendants
                        conditionants -> union(cd.getConditioner());
                }
                conditioners -> union(conditionants);
                conditioners.remove(universal); // - {Object}
                if (conditioners.size() == oldSize) {
                        changes = false; // exit loop
                }
        }
        conditioners.add(universal); // add universal class finally
}
```

La función se apoya en la definición del método `getConditioner` de la clase `ClassDef` del metamodelo MOON, que devuelve los ancestros y proveedores de una clase. Por otro lado también se utiliza el método `getDescendants` en dicha clase, que consulta todos los descendientes de una clase.

D.2. EquivalentEntities

Descripción: se define el conjunto de entidades equivalentes a una entidad dada como:

- Si es una entidad interna (declaración local (`Local`)), la única entidad equivalentes es la propia entidad.
- Si es una entidad de signatura (atributo (`Attribute`) o argumento formal (`Formal-Argument`)):
  - Si es una propiedad se añaden las propiedades equivalentes.
  - Si es el i-ésimo argumento formal de un método, se recopilan los argumentos equivalentes por posición en todos los métodos equivalentes en la jerarquía de herencia.

Código D.2.2: Función `EquivalentEntities`

```
namespace Moon::Function;

activity EquivalentEntities(in entity : Entity) : List<Entity> {
```

```
        List<Entity> equiv = new List<Entity>();

        if (entity instanceof LocalDec){
                equiv.add(entity);
        }
        else if (entity instanceof SignatureEntity){
                if (entity instanceof Property) {
                        List<Property> properties = EquivalentProperties((Property)
                            entity);
                        for (Property p : properties){
                                equiv.add((Entity) p);
                        }
                }
                else if (entity instanceof FormalArgument){
                        Method md = ((FormalArgument) entity).getMethDec();
                        List<Property> properties = EquivalentProperties(md);
                        Integer index = md.getIndexFormalArg((FormalArgument) entity);
                        for (Property p : properties) {
                                if (p instanceof Method){
                                        Method temp = (Method) p;
                                        List<FormalArgument> arguments = temp.
                                            getFormalArgument();
                                        if (arguments.size() > index){
                                                equiv.add(arguments.toSequence()[index
                                                    ]);
                                        }
                                }
                        }
                }
        }
        return equiv;
}
```

Dicha función se apoya en la función auxiliar **EquivalentProperties** definida tal y como se muestra en el Código D.2.3, a su vez utilizando las funciones **EquivalentUpProperties** (ver Código D.2.4) y **EquivalentDownProperties** (ver Código D.2.5).

Código D.2.3: Función `EquivalentProperties`

```
namespace Moon::Function;

activity EquivalentProperties(in property : Property) : List<Property> {
        List<Property> equiv = new List<Property>();

        // Proper descendants
        equiv.addAll(EquivalentUpProperties(property));

        // Proper ancestors
        equiv.addAll(EquivalentDownProperties(property));

        return equiv;
}
```

Estas funciones recolectan las propiedades equivalentes, en este caso los métodos, en sentido ascendente o descendente en la jerarquía de herencia, apoyándose en los métodos para la navegación en la jerarquía de herencia proporcionados por el metamodelo MOON (`getProperAncestors` y `getProperDescendants` respectivamente).

Código D.2.4: Función `EquivalentUpProperties`

```
namespace Moon::Function;

activity EquivalentUpProperties(in property : Property) : List<Property> {
        List<Property> equiv = new List<Property>();
        if (property instanceof Method){
                Method md = (Method) property;
                // Proper ancestors
                List<ClassDef> properDescendants = md.getClassDef().
                    getProperDescendants();
                for (ClassDef descendants : properDescendants){
                        List<Method> list = descendants.getMethods();
                        for (Method other : list){
                                if (other.hasSameSignature(md)){
                                        equiv.add(other);
                                }
                        }
                }
        }
        return equiv;
}
```

Código D.2.5: Función `EquivalentDownProperties`

```
namespace Moon::Function;

activity EquivalentDownProperties(in property : Property) : List<Property> {
        List<Property> equiv = new List<Property>();
        if (property instanceof Method){
                Method md = (Method) property;
                // Proper ancestors
                List<ClassDef> properAncestors = md.getClassDef().getProperAncestors();
                for (ClassDef ancestor : properAncestors){
                        List<Method> list = ancestor.getMethods();
                        for (Method other : list){
                                if (other.hasSameSignature(md)){
                                        equiv.add(other);
                                }
                        }
                }
        }
        return equiv;
}
```

D.3. GenericDependant

Descripción: dada la clase objetivo, la entidad de entrada determina el conjunto de entidades en GDD (genérico dependientes directas) y GDI (genérico dependientes indirectas), atendiendo a un conjunto de reglas de construcción (o pasos). El algoritmo es de punto fijo, donde la condición de finalización se establece como la no modificación de ninguno de los dos conjuntos en una de sus iteraciones.

Como se indica en [Crespo, 2000], la numeración de dichas reglas no implica un orden sino una referencia, que ha sido mantenida en la especificación en ALF, para facilitar su lectura respecto a la definición original. Aunque se debe señalar que en algunos casos, las referencias

o pasos han sido incluidas como partes de otros, como ocurre con las reglas 8 y 11, así como la eliminación de la regla 5, dado que en el metamodelo ya no se utiliza por separado el concepto de función y su resultado, sino que se utiliza sólo el concepto de `Result` ligado a la declaración estática de la función.

Por motivos de brevedad, se recomienda una consulta a [Crespo, 2000, p. 158] para una descripción más detallada del algoritmo.

Código D.3.6: Función `GenericDependant`

```
namespace Moon::Function;

public import Moon::GenericDependantUtilities::GetCStar;
public import Moon::GenericDependantUtilities::GetEntitiesCStar;
public import Moon::GenericDependantUtilities::Step1;
public import Moon::GenericDependantUtilities::Step2;
public import Moon::GenericDependantUtilities::Step3;
public import Moon::GenericDependantUtilities::Step4;
public import Moon::GenericDependantUtilities::Step5;
public import Moon::GenericDependantUtilities::Step6;
public import Moon::GenericDependantUtilities::Step7;
public import Moon::GenericDependantUtilities::Step9;
public import Moon::GenericDependantUtilities::Step10;

activity GenericDependant(in targetEntity : Entity, inout gdi : List<Entity>) : List<
    Entity>{
        // Aux. sets, initializations...
        List<Entity> cStarGdd = new List<Entity>();
        List<Entity> cStarGdi = new List<Entity>();
        // genGdd and genGdi collected in Step9OptionA and Step10OptionA
        List<Entity> genGdd = new List<Entity>();
        List<Entity> genGdi = new List<Entity>();
        // Universe of classes...
        List<ClassDef> universe = new List<ClassDef>();
        universe.addAll(ParameterizeUniverse(targetEntity.getClassDef()).toSequence()
            -> select c (c.isSourceAvailable()));
        // Entities...
        List<Entity> entitiesOfUniverse = new List<Entity>();
        for (ClassDef cd : universe) {
                entitiesOfUniverse.addAll(cd.getEntities());
        }
        List<ClassDef> cStar = GetCStar(targetEntity.getType().getClassDef());
        List<Entity> entitiesCStar = GetEntitiesCStar(targetEntity.getType().
            getClassDef());
        ClassDef classDef = targetEntity.getClassDef();
        List<Entity> gdd = new List<Entity>();

        ////////////////////////////////////////////////
        // Algorithm
        int gddSize = gdd.size();
        int gdiSize = gdi.size();

        // First rule, initialize with target entity (step0)
        gdd.add(targetEntity);
        // Added "self" entity of the current class where is declared the entity
        gdi.add(classDef.getSelfEntity());

        int newGddSize = gdd.size();
        int newGdiSize = gdi.size();
        // while the number of entities is changing in gdd or gdi sets...
        while(newGddSize != gddSize || newGdiSize != gdiSize) {
```

```
                gddSize = gdd.size();
                gdiSize = gdi.size();

                Step1(entitiesCStar, entitiesOfUniverse, genGdd, genGdi, gdi);
                // Step 2
                for (ClassDef cd : universe) {
                        Step2(entitiesOfUniverse, cd, gdd);
                        Step2(entitiesOfUniverse, cd, gdi);
                }
                // Step 3
                for (Entity ei : entitiesOfUniverse) {
                        Step3(cStar, ei, genGdd, genGdi, gdd);
                        Step3(cStar, ei, genGdd, genGdi, gdi);
                }
                // Step 4
                Step4(gdd);
                Step4(gdi);
                // Step5();      not included in current metamodel
                Step6(universe, gdd, gdi);
                Step7(universe, gdd, gdi);
                // Step 8 embedded in Step6 and Step7
                Step9(universe,entitiesCStar,cStarGdd,cStarGdi,genGdd,genGdi,gdd,gdi);
                Step10(universe,entitiesCStar,cStarGdd,cStarGdi,genGdd,genGdi,gdd,gdi);
                // Step 11 embedded in step 9
                newGddSize = gdd.size();
                newGdiSize = gdi.size();
        }
        return gdd;
}
```

A continuación se desglosan cada uno de los pasos del algoritmo, así como el conjunto de funciones auxiliares necesarios para su ejecución, agrupadas en el mismo espacio de nombres en su declaración en ALF.

La actividad `Step1` recopila la auto-referencia en el conjunto GDI, de aquellas clases que contienen una entidad que es genérico dependiente indirecta de la entidad de entrada, y aún siendo su clase genérica, dicha clase no tiene una entidad con tipo variable que participe en una asociación con una entidad generico dependiente.

Código D.3.7: Función `Step1`

```
namespace Moon::GenericDependantUtilities;

activity Step1(in entitiesCStar : List<Entity>, in entitiesUniverse : List<Entity>,
                        in genGdd : List<Entity>, in genGdi : List<Entity>,
                        inout gdi : List<Entity>) {
        (entitiesUniverse -> select ei (gdi->includes(ei) && !entitiesCStar
        -> includes(ei) && !Gen(ei.getClassDef(), genGdd, genGdi))).toSequence()
                -> iterate e (gdi.add(e.getType().getClassDef().getSelfEntity()));
}
```

La actividad `Step2` recopila al correspondiente conjunto todas aquellas entidades cuyo tipo estático viene determinado por la clase cuya auto-referencia está ya contenida en el conjunto actual (ya sea GDD o GDI). La entidad a añadir no puede ser además la referencia paterna, ni su tipo estar definido por un parámetro formal.

Código D.3.8: Función `Step2`

```
namespace Moon::GenericDependantUtilities;

activity Step2(in entitiesUniverse : List<Entity>, in classDef : ClassDef, inout gdx :
    List<Entity>) {
                if (gdx.includes(classDef.getSelfEntity())) {
                        (entitiesUniverse ->
                select ei (((Entity) ei).getType().getClassDef() == classDef
                                        && !(ei instanceof Super)
                                        && !(((Entity)ei).getType() instanceof
                                            FormalPar))
                                        ).toSequence() -> iterate e (gdx.add(e));
                }
}
```

La actividad `Step3`, recopila la auto-referencia de la clase al conjunto GDI, para toda entidad de las clases del universo, si dicha entidad es genérico dependiente de la entidad de entrada y aún siendo su clase genérica, dicha clase no tiene una entidad con tipo variable que participe en una asociación con una entidad genérico dependiente.

Código D.3.9: Función `Step3`

```
namespace Moon::GenericDependantUtilities;

activity Step3(in cStar : List<ClassDef>, in entity : Entity,
                                in genGdd : List<Entity>, in genGdi : List<Entity>,
                inout gdx : List<Entity>) {
        if (gdx.includes(entity)
                        && !cStar.includes(entity.getType().getClassDef())
                        && !Gen(entity.getType().getClassDef(), genGdi, genGdi)
                        && !(entity instanceof Super)) { // not a super entity
                gdx.add(entity.getType().getClassDef().getSelfEntity());
        }
}
```

La actividad `Step4` añade a partir de toda entidad del conjunto actual (GDD o GDI) el conjunto de entidades equivalentes, si no se trata de la auto-referencia, y el conjunto de entidades equivalentes hacia abajo en caso contrario.

Código D.3.10: Función `Step4`

```
namespace Moon::GenericDependantUtilities;

public import Moon::Function::EquivalentEntities;
public import Moon::Function::EquivalentDownEntities;

activity Step4(inout gdx : List<Entity>) {
        List<Entity> initial = new List<Entity>();
        initial.addAll(gdx);
        for (Entity ei : initial) {
                if (! (ei instanceof Self)) {
                        EquivalentEntities(ei).toSequence() -> iterate entity (gdx.add(
                            entity));
                }
                else {
                        EquivalentDownEntities(ei).toSequence() -> iterate entity (gdx.
                            add(entity));
                }
        }
}
```

La actividad `Step6` recopila las entidades genérico dependientes a través de asociaciones libres entre entidades en asignaciones (*e.g.* $e_1 := e_2$).

Código D.3.11: Función `Step6`

```
namespace Moon::GenericDependantUtilities;

activity Step6(in universe : List<ClassDef>, inout gdd : List<Entity>, inout gdi: List<
    Entity>) {
  for (ClassDef cd : universe) {
        for (Method md : cd.getMethods()) {
        List<Instr> list = md.getFlattenedInstructions();
        for (Instr instr : list) {
                        if (instr instanceof AssignmentInstr) {
                                AssignmentInstr asigInstr = (AssignmentInstr) instr;
                                Expr left = asigInstr.getLeftSide();
                                Expr right = asigInstr.getRightSide();
                                if (left instanceof CallExpr
                                                && right instanceof CallExpr
                                                && ((CallExpr) left).getLeftSide() ==
                                                    null
                                                && ((CallExpr) right).getLeftSide() ==
                                                    null) {
                                        Entity leftEntity = ((CallExpr) left).getTarget
                                            ();
                                        Entity rightEntity = ((CallExpr) right).
                                            getTarget();
                                        // step 8 embeded if right is a Result...

                                        Step6-7substep(leftEntity, rightEntity, gdd);
                                        Step6-7substep(leftEntity, rightEntity, gdi);

                                        Step6-7substep(rightEntity, leftEntity, gdd);
                                        Step6-7substep(rightEntity, leftEntity, gdi);

                                        // step 8 embeded here
                                        if (rightEntity instanceof Result) {
                                                List<Expr> realArguments = ((CallExpr)
                                                    right).getRealArguments();
                                        Method mdInvoked = ((Result) rightEntity).
                                            getFunctionDec();
                                                Step7CheckArguments(mdInvoked,
                                                    realArguments, gdd, gdi);
                                        }
                                }
                        }
                }
        }
  }
}
```

La actividad `Step6-7substep`, añade la entidad del lado derecho de una asignación si la entidad del lado izquierdo pertenece a uno de los conjuntos de entidades genérico dependientes.

Código D.3.12: Función `Step6-7substep`

```
namespace Moon::GenericDependantUtilities;

activity Step6-7substep(in leftEntity: Entity, in rightEntity : Entity, inout gdx :
    List<Entity>) {
```

```
        if (gdx.includes(leftEntity)) {
                gdx.add(rightEntity);
        }
}
```

La actividad `Step6bis` repite el proceso realizado en `Step6`, pero teniendo en cuenta que estas asignaciones libres se pueden producir en la declaración de atributos con inicializadores.

Código D.3.13: Función `Step6bis`

```
namespace Moon::GenericDependantUtilities;

activity Step6Bis(in universe : List<ClassDef>, inout gdd : List<Entity>, inout gdi:
    List<Entity>) {
  for (ClassDef cd : universe) {
        for (Attribute ad : cd.getAttributes()) {
          if (ad.hasInitializer()) {
                Expr right = ad.getInitializer();
                // free association between e1 := e2 ?
                if (right instanceof CallExpr
                                && ((CallExpr) right).getLeftSide() == null) {
                  Entity leftEntity = ad;
                  Entity rightEntity = ((CallExpr) right).getTarget();
                  // step 8 embeded if right is a Result...
                  Step6-7substep(leftEntity, rightEntity, gdd);
                  Step6-7substep(leftEntity, rightEntity, gdi);

                  // step 8 embedded here
                  if (rightEntity instanceof Result) {
                        List<Expr> realArguments = ((CallExpr) right).getRealArguments
                            ();
                        Method mdInvoked = ((Result) rightEntity).getFunctionDec();
                        Step7CheckArguments(mdInvoked, realArguments, gdd, gdi);
                  }
                }
          }
        }
  }
}
```

La actividad `Step7` recopila las entidades genérico dependientes a través de asociaciones libres entre entidades como resultado del paso de argumentos en la invocación a métodos (*e.g.* `arg` como sustitución a e_3 en la invocación a $m(..., e_3, ...)$).

Código D.3.14: Función `Step7`

```
namespace Moon::GenericDependantUtilities;

activity Step7(in universe : List<ClassDef>, inout gdd : List<Entity>, inout gdi: List<
    Entity>) {
  for (ClassDef cd : universe) {
        for (Method md : cd.getMethods()) {
          List<Instr> list = md.getFlattenedInstructions();
          for (Instr instr : list) {
                if (instr instanceof CallInstr) {
                  CallInstr callInstr = (CallInstr) instr;
                  // if free association...
                  if (callInstr.getLeftSide() == null) {
```

```
                        Method mdInvoked = callInstr.getRoutineDec();
                        List<Expr> arguments = callInstr.getRealArguments();
                        Step7CheckArguments(mdInvoked, arguments, gdd, gdi);
                    }
                }
            }
        }
    }
}
```

La actividad `Step7CheckArguments`, es una función auxiliar utilizada en `Step7`, para comprobar la correspondencia entre los argumentos actuales y formales.

Código D.3.15: Función `Step7CheckArguments`

```
namespace Moon::GenericDependantUtilities;

activity Step7CheckArguments(in method : Method, in arguments : List<Expr>, inout gdd :
     List<Entity>, inout gdi : List<Entity>) {
  Integer index = 0;
  for (Expr argument : arguments) {
        if (argument instanceof CallExpr) {
                CallExpr argumentExpr = (CallExpr) argument;
                // if free association
                if (argumentExpr.getLeftSide() == null) {
                        Entity formalArg = method.getFormalArgument().at(index);
                        Step6_7substep(argumentExpr.getTarget(), formalArg, gdd);
                        Step6_7substep(argumentExpr.getTarget(), formalArg, gdi);
                }
        }
        index++;
  }
}
```

La actividad `Step9` recopila las entidades genérico dependientes a través de asociaciones atadas entre entidades, en asignaciones de la forma $e_1 := a.e_2$ donde e_1 y e_2 están atadas a través de la entidad a.

Código D.3.16: Función `Step9`

```
namespace Moon::GenericDependantUtilities;

activity Step9(in universe : List<ClassDef>, in entitiesCStar : List<Entity>,
        inout cStarGdd : List<Entity>, inout cStarGdi : List<Entity>,
        inout genGdd : List<Entity>, inout genGdi: List<Entity>,
        inout gdd : List<Entity>, inout gdi: List<Entity>) {

  for (ClassDef cd : universe) {
        for (Method md : cd.getMethods()) {
          List<Instr> list = md.getFlattenedInstructions();
          for (Instr instr : list) {
                if (instr instanceof AssignmentInstr) {
                  AssignmentInstr asigInstr = (AssignmentInstr) instr;
                  Expr left = asigInstr.getLeftSide();
                  Expr right = asigInstr.getRightSide();

                  // not free association between e1 := a.e2 ?
                  if (left instanceof CallExpr
                                && right instanceof CallExpr
                                && ((CallExpr) left).getLeftSide() == null
```

```
                                && ((CallExpr) right).getLeftSide() != null) {
                                // not free get init values

                        Entity leftEntity = ((CallExpr) left).getTarget();
                        Entity rightEntity = ((CallExpr) right).getTarget();
                        Expr exprAux = ((CallExpr) right).getLeftSide();
                        Entity bondEntity = (Entity) null;

            if (exprAux instanceof CallExpr) {
                      bondEntity = ((CallExpr) exprAux).getTarget();
                        }
                        // check the three cases A-B-C
                        if (InGDX(leftEntity, gdd, gdi)) { // Option A
                      Step9OptionA(leftEntity, rightEntity, bondEntity, entitiesCStar,
                                                        cStarGdd, cStarGdi, genGdd,
                                                            genGdi, gdd, gdi);
                        } else if (InGDX(rightEntity, gdd, gdi)) { // Option B
                          Step9OptionB(leftEntity, rightEntity, bondEntity, gdd, gdi);
                        } else { // Option C
                          Step9OptionC(leftEntity, rightEntity, bondEntity,
                              entitiesCStar,
                                                        cStarGdd, cStarGdi, genGdd,
                                                            genGdi, gdd, gdi);
                        }
                        // end of rule 9

                        // modification to embed the step 11
                        // repeating step 10
                        // with e1 = a.e2(... , e3, ....)
                        // process a.e2(..., e3, ...) as function result
                        ProcessExprAsFunctionResult(right, entitiesCStar,
                                                    cStarGdd, cStarGdi, genGdd, genGdi, gdd
                                                        , gdi);
                    }
                    if (left instanceof CallExpr) {
                          ProcessExprAsFunctionResult(left, entitiesCStar,
                                                    cStarGdd, cStarGdi, genGdd, genGdi, gdd
                                                        , gdi);
                    }
                } // if it is an assignment
              } // for each instr
            } // for each method
  } // for each class in universe
}
```

La actividad `Step9OptionA` procesa la entidad atada y la entidad del lado derecho de la asignación, siempre y cuando la entidad del lado izquierdo sea genérico dependiente.

Código D.3.17: Función `Step9OptionA`

```
namespace Moon::GenericDependantUtilities;

public import Moon::Predicate::IsRecursivePositive;

activity Step9OptionA(in e1: Entity, in e2 : Entity, in a : Entity,
                        in entitiesCStar : List<Entity>,
                        inout cStarGdd : List<Entity>, inout cStarGdi : List<Entity>,
                        inout genGdd : List<Entity>, inout genGdi: List<Entity>,
                        inout gdd : List<Entity>, inout gdi: List<Entity>) {

  // e1 = a.e2;
```

```
  if (IsRecursivePositive(e2)) {
        // e2 in the proper gdx
        AddEntityInProperGDX(e1, e2, gdd, gdi);
  } else { // not recursive
    gdi.add(a);
  }

  if (!entitiesCStar.includes(e2) && e2.getType().isComplete()) {
        // e2 in the proper gdx
        AddEntityInProperGDX(e1, e2, gdd, gdi);
 } else if (!e2.getType().isComplete()) { // in CStar but is Var
        // toAdd in the proper gdx if check belongs to one or other GDX
        if (gdd.includes(e1)) {
      genGdd.add(e2);
        } else {
          genGdi.add(e2);
        }
  } else if (entitiesCStar.includes(e2)) {
        // toAdd in the proper gdx if check belongs to one or other GDX
        if (gdd.includes(e1)) {
          cStarGdd.add(e2);
        } else {
          cStarGdi.add(e2);
        }
  }
}
```

La actividad `Step9OptionB` procesa la entidad atada y la entidad del lado izquierdo de la asignación, siempre y cuando la entidad asignada al lado derecho sea genérico dependiente.

Código D.3.18: Función `Step9OptionB`

```
namespace Moon::GenericDependantUtilities;

public import Moon::Predicate::IsRecursivePositive;

activity Step9OptionB(in e1: Entity, in e2 : Entity, in a : Entity,
                              inout gdd : List<Entity>, inout gdi: List<Entity>) {
        // e1 = a.e2;
        if (IsRecursivePositive(e2)) {
                // a in the proper gdx
                AddEntityInProperGDX(e2, a, gdd, gdi);
        } else { // not recursive positive
                gdi.add(a);
        }
        // e1 in the proper gdx
        AddEntityInProperGDX(e2, e1, gdd, gdi);
}
```

La actividad `Step9OptionC` procesa la entidad del lado izquierdo de la asignación si la entidad a través de la que están atadas el lado izquierdo y derecho de la asignación, es genérico dependiente y la entidad del lado derecho de la asignación no introduce recursividad positiva.

Código D.3.19: Función `Step9OptionC`

```
namespace Moon::GenericDependantUtilities;

public import Moon::Predicate::IsRecursivePositive;
```

```
activity Step9OptionC(in e1: Entity, in e2 : Entity, in a : Entity,
                      in entitiesCStar : List<Entity>,
                      in cStarGdd : List<Entity>, in cStarGdi : List<Entity>,
                      in genGdd : List<Entity>, in genGdi: List<Entity>,
                      inout gdd : List<Entity>, inout gdi: List<Entity>) {
  if (a != null && InGDX(a, gdd, gdi) && IsRecursivePositive(a)) {
        // e1 in the proper gdx
        AddEntityInProperGDX(a, e1, gdd, gdi);
  } else if ( (gdi.includes(a) && entitiesCStar.includes(e2))
                || !e2.getType().isComplete()) {
        if (genGdd.includes(e1)
                || cStarGdd.includes(e1)) {
                gdd.add(e1);
        } else if (genGdi.includes(e1)
                        || cStarGdi.includes(e1)) {
                gdi.add(e1);
        }
  }
}
```

La actividad `Step10` recopila las entidades genérico dependientes a través de asociaciones atadas entre entidades, como resultado del paso de argumentos en la invocación $a.m(..., arg, ...)$ a métodos con signatura $a.m(..., e_3, ...)$ donde el argumento actual arg y el argumento formal e_3 están atados a través de la entidad a.

Código D.3.20: Función `Step10`

```
namespace Moon::GenericDependantUtilities;

activity Step10(in universe : List<ClassDef>,
                      in entitiesCStar : List<Entity>,
                      inout cStarGdd : List<Entity>, inout cStarGdi : List<Entity>,
                      inout genGdd : List<Entity>, inout genGdi: List<Entity>,
                      inout gdd : List<Entity>, inout gdi : List<Entity>) {

  for (ClassDef cd : universe) {
        for (Method md : cd.getMethods()) {
          List<Instr> list = md.getFlattenedInstructions();
          for (Instr instr : list) {
                if (instr instanceof CallInstr) { // call instructions
                  CallInstr callInstr = (CallInstr) instr;
                  // Forward reading...
                  if (callInstr.getLeftSide() != null) {
                        Expr boundSide = callInstr.getLeftSide();
                        Method mdInvoked = callInstr.getRoutineDec();
                        List<Expr> arguments = callInstr.getRealArguments();

                        // if not free association...
                        ProcessLeftExpressions(boundSide, mdInvoked, arguments,
                                        entitiesCStar, cStarGdd, cStarGdi, genGdd,
                                            genGdi, gdd, gdi);
                  }
            }
          }
    }
  }
}
```

La actividad auxiliar `Step10CheckArguments` procesa los argumentos actuales junto con los correspondientes argumentos formales, comprobando los distintos casos (`Step10OptionA`, `Step10OptionB` y `Step10OptionC`).

Código D.3.21: Función `Step10CheckArguments`

```
namespace Moon::GenericDependantUtilities;

activity Step10CheckArguments(in method : Method, in arguments : List<Expr>, in entity
    : Entity,
        in entitiesCStar : List<Entity>, inout cStarGdd : List<Entity>,
        inout cStarGdi : List<Entity>, inout genGdd : List<Entity>,
        inout genGdi: List<Entity>,     inout gdd : List<Entity>,
        inout gdi : List<Entity>) {
  Integer index = 0;
  for (Expr argument : arguments) {
        if (argument instanceof CallExpr) {
          CallExpr argumentExpr = (CallExpr) argument;
          // if free association
          if (argumentExpr.getLeftSide() == null) {
                Entity leftEntity = argumentExpr.getTarget();
                Entity rightEntity = method.getFormalArgument().at(index);
                // check the three cases A-B-C
                if (InGDX(leftEntity, gdd, gdi)) { // Option A
                        Step10OptionA(leftEntity, rightEntity, entity,
                entitiesCStar, cStarGdd, cStarGdi,
                genGdd, genGdi, gdd, gdi);
                } else if (InGDX(rightEntity, gdd, gdi)) { // Option B
                        Step10OptionB(leftEntity, rightEntity, entity, gdd, gdi);
                } else { // Option C
                        Step10OptionC(leftEntity, rightEntity, entity,
                entitiesCStar, cStarGdd, cStarGdi, genGdd, genGdi, gdd, gdi);
                }
          }
    }
        index++;
  }
}
```

La actividad `Step10OptionA` procesa la entidad atada y el argumento formal correspondiente si el argumento actual pertenece al conjunto de entidades genérico dependientes.

Código D.3.22: Función `Step10OptionA`

```
namespace Moon::GenericDependantUtilities;

public import Moon::Predicate::IsRecursiveNegative;

activity Step10OptionA(in realArgument : Entity, in formalArgument : Entity,
             in bound : Entity, in entitiesCStar : List<Entity>,
             inout cStarGdd : List<Entity>, inout cStarGdi : List<Entity>,
             inout genGdd : List<Entity>, inout genGdi: List<Entity>,
             inout gdd : List<Entity>, inout gdi : List<Entity>){
  if (InGDX(realArgument, gdd, gdi)) {
        if (IsRecursiveNegative(formalArgument)) {
           // e2 in the proper gdx
                AddEntityInProperGDX(realArgument, bound, gdd, gdi);
        } else { // not recursive
           // if the entity is not parametric..
                if (!((ClassType)bound.getType()).isParametric() ||
```

```
                        (((ClassType)bound.getType()).isParametric() &&
                        !(formalArgument.getType() instanceof FormalPar))){
                  gdi.add(bound);
                }
        }

        if (!entitiesCStar.includes(formalArgument)
                && formalArgument.getType().isComplete()) {
           // formalArgument in the proper gdx
                AddEntityInProperGDX(realArgument, formalArgument, gdd, gdi);
        } else if (!formalArgument.getType().isComplete()) { // in CStar but
           // toAdd in the proper gdx if check belongs to one or other GDX
           if (gdd.includes(realArgument)) {
            genGdd.add(formalArgument);
           } else {
                genGdi.add(formalArgument);
           }
        } else if (entitiesCStar.includes(formalArgument)) {
          // toAdd in the proper gdx if check belongs to one or other GDX
          if (gdd.includes(realArgument)) {
                cStarGdd.add(formalArgument);
          } else {
                cStarGdi.add(formalArgument);
          }
        }
  }
}
```

La actividad `Step10OptionB` procesa la entidad atada y el argumento actual correspondiente si el argumento formal pertenece al conjunto de entidades genérico dependientes.

Código D.3.23: Función `Step10OptionB`

```
namespace Moon::GenericDependantUtilities;

public import Moon::Predicate::IsRecursiveNegative;

activity Step10OptionB(in realArgument : Entity, in formalArgument : Entity,
        in bound : Entity, inout gdd : List<Entity>, inout gdi : List<Entity>){
  if (InGDX(formalArgument, gdd, gdi)) {
        AddEntityInProperGDX(formalArgument, realArgument, gdd, gdi);
        if (IsRecursiveNegative(formalArgument)) {
                // a in the proper gdx
                AddEntityInProperGDX(formalArgument, bound, gdd, gdi);
        } else { // not recursive negative
                gdi.add(bound);
        }
  }
}
```

La actividad `Step10OptionC` procesa el argumento actual si la entidad atada está en el conjunto de entidades genérico dependientes y el argumento formal no tiene recursividad negativa.

Código D.3.24: Función `Step10`

```
namespace Moon::GenericDependantUtilities;

public import Moon::Predicate::IsRecursiveNegative;
```

```
activity Step10OptionC(in realArgument : Entity, in formalArgument : Entity,
                       in bound : Entity, in entitiesCStar : List<Entity>,
                       in cStarGdd : List<Entity>, in cStarGdi : List<Entity>,
                       in genGdd : List<Entity>, in genGdi: List<Entity>,
                       inout gdd : List<Entity>, inout gdi : List<Entity>){
  if (bound != null && InGDX(bound,gdd,gdi)
                && IsRecursiveNegative(formalArgument)) {
    // e1 in the proper gdx
        AddEntityInProperGDX(bound, realArgument, gdd, gdi);
  } else if (gdi.includes(bound)
                       && (entitiesCStar.includes(formalArgument))
                       || !formalArgument.getType().isComplete()) {
      if (genGdd.includes(formalArgument)
                       || cStarGdd.includes(formalArgument)) {
        gdd.add(realArgument);
      } else if (genGdi.includes(formalArgument)
                       || cStarGdi.includes(formalArgument)) {
                gdi.add(realArgument);
          }
  }
}
```

A continuación se desglosan aquellas funciones auxiliares necesarias, para la ejecución correcta del algoritmo.

La actividad `AddEntityInProperGDX`, añade la entidad al correspondiente conjunto en el que no estaba previamente incluida.

Código D.3.25: Función `AddEntityInProperGDX`

```
namespace Moon::GenericDependantUtilities;

activity AddEntityInProperGDX(in entityCheck : Entity, in entityAdd : Entity, in gdd :
    List<Entity>, in gdi : List<Entity>) {
        if (!gdd.includes(entityCheck)) {
                gdd.add(entityAdd);
        }
        else {
                gdi.add(entityAdd);
        }
}
```

La actividad `Gen`, comprueba si la clase es genérica y tiene una entidad con un tipo variable que participa en una asociación con una entidad genérico dependiente.

Código D.3.26: Función `Gen`

```
namespace Moon::GenericDependantUtilities;

activity Gen(in classDef : ClassDef, in genGdd : List<Entity>,
                in genGdi : List<Entity>) : Boolean {
        if (classDef.isGeneric()) {
                List<Entity> list = classDef.getEntities();
                for (Entity entity : list) {
                        if (entity.getType() instanceof FormalPar) {
                                if (genGdd.includes(entity)
                                        || genGdi.includes(entity)) {
                                        return true;
                                }
                        }
```

```
                }
        }
        return false;
}
```

La actividad `GetCStar` devuelve el conjunto de ancestros y descendientes de una clase.

Código D.3.27: Función `GetCStar`

```
namespace Moon::GenericDependantUtilities;

activity GetCStar(in classDef : ClassDef) : List<ClassDef> {
        List<ClassDef> total = new List<ClassDef>();
        total.addAll(classDef.getAncestors());
        total.addAll(classDef.getDescendants());
        return total;
}
```

La actividad `GetEntitiesCStar` devuelve el conjunto de entidades en los ancestros y descendientes de una clase.

Código D.3.28: Función `GetEntitiesCStar`

```
namespace Moon::GenericDependantUtilities;

public import Moon::Function::Entities;

activity GetEntitiesCStar(in classDef : ClassDef) : List<Entity> {
        List<Entity> total = new List<Entity>();
        List<ClassDef> universe = new List<ClassDef>();
        universe.addAll(GetCStar(classDef));

        for (ClassDef cdAux : universe) {
                List<Entity> listEntities = Entities(cdAux);
                for (Entity entity : listEntities) {
                        if (entity.getClassDef() != null &&
                                entity.getClassDef().isSourceAvailable() &&
                                !total.includes(entity)) {
                          total.add(entity);
                        }
                }
        }
        return total;
}
```

La actividad `InGDX` consulta si la entidad es genérico dependiente, ya sea directa o indirectamente.

Código D.3.29: Función `InGDX`

```
namespace Moon::GenericDependantUtilities;

activity InGDX(in entity : Entity, in gdd : List<Entity>, in gdi : List<Entity>) :
    Boolean {
        return gdd.includes(entity) || gdi.includes(entity);
}
```

La actividad `ProcessExprAsFunctionResult`, procesa una invocación a una función extrayendo el método invocado con sus argumentos formales, sus argumentos actuales y la entidad a la que está ligada dicha invocación, completando el proceso realizado en `Step9`.

Código D.3.30: Función `ProcessExprAsFunctionResult`

```
namespace Moon::GenericDependantUtilities;

activity ProcessExprAsFunctionResult(in expr : Expr,
                        in entitiesCStar : List<Entity>,
                        inout cStarGdd : List<Entity>, inout cStarGdi : List<Entity>,
                        inout genGdd : List<Entity>,    inout genGdi: List<Entity>,
                        inout gdd : List<Entity>, inout gdi : List<Entity>) {

        let mdInvoked : Method  = new Method();
        List<Expr> arguments = new List<Expr>();

        if (expr instanceof CallExpr) {
                CallExpr callExpr = (CallExpr) expr;
                if (callExpr.getTarget() instanceof Result) {
                        Expr boundSide = callExpr.getLeftSide();
                        mdInvoked = ((Result) callExpr.getTarget()).getFunctionDec();
                        arguments.addAll(callExpr.getRealArguments());
                        ProcessLeftExpressions(boundSide, mdInvoked, arguments,
                            entitiesCStar, cStarGdd, cStarGdi, genGdd, genGdi, gdd, gdi
                            );
                }
                else {
                        Expr boundSide = callExpr.getLeftSide();
                        if (boundSide != null
                                        && boundSide instanceof CallExpr
                                        && ((CallExpr) boundSide).getTarget()
                                            instanceof Result) {
                                mdInvoked = ((Result) ((CallExpr) boundSide).getTarget
                                    ()).getFunctionDec();
                                arguments.addAll(callExpr.getRealArguments());
                                ProcessLeftExpressions(boundSide, mdInvoked, arguments,
                                     entitiesCStar, cStarGdd, cStarGdi, genGdd, genGdi,
                                     gdd, gdi);
                        }
                }
        }
}
```

La actividad `ProcessLeftExpressions` procesa iterativamente las invocaciones a métodos al lado izquierdo de la expresión inicial.

Código D.3.31: Función `ProcessLeftExpressions`

```
namespace Moon::GenericDependantUtilities;

activity ProcessLeftExpressions(in expr : Expr, in method : Method,
            in arguments : List<Expr>, in entitiesCStar : List<Entity>,
            inout cStarGdd : List<Entity>, inout cStarGdi : List<Entity>,
            inout genGdd : List<Entity>,        inout genGdi: List<Entity>,
            inout gdd : List<Entity>, inout gdi : List<Entity>) {
        Expr localExpr = expr;
        Expr exprAux = localExpr;
        Method mdInvoked = method;
        List<Expr> argumentsAux = arguments;
```

```
        Boolean flag = true;
        while (flag) {
                if (localExpr instanceof CallExpr) {
                        Entity boundEntity = ((CallExpr) localExpr).getTarget();
                        Step10CheckArguments(mdInvoked, argumentsAux, boundEntity,
                    entitiesCStar, cStarGdd, cStarGdi, genGdd, genGdi, gdd, gdi);
                }
                // if there is left side to process in next iterations...
                if (localExpr instanceof CallExpr
                                && ((CallExpr) localExpr).getLeftSide() != null) {
                        localExpr = ((CallExpr) localExpr).getLeftSide();
                        exprAux = ((CallExpr) localExpr).getRightSide();
                        if ( ((CallExpr) exprAux).getTarget() instanceof Result) {
                                mdInvoked = ((Result) ((CallExpr) exprAux).getTarget())
                                    .getFunctionDec();
                                argumentsAux = ((CallExpr) exprAux).getRealArguments();
                        }
                } else {
                        flag = false; // Exit while loop
                }
        }
}
```

Adicionalmente se incluye una nueva función principal, no vinculada exclusivamente a la resolución del algoritmo de detección de entidades genérico dependientes, como es la función **`EquivalentDownEntities`**, que devuelve el conjunto de entidades equivalentes hacia abajo en el árbol de herencia para lo que debe buscarlas en los ancestros. Utiliza de manera auxiliar la función **`EquivalentDownProperties`** (ver Código D.2.5).

Código D.3.32: Función `EquivalentDownEntities`

```
namespace Moon::Function;

activity EquivalentDownEntities(in entity : Entity) : List<Entity> {
        List<Entity> equiv = new List<Entity>();
        if (entity instanceof Local) {
                equiv.add(entity);
        }
        else if (entity instanceof SignatureEntity) {
                if (entity instanceof Property) {
                        List<Property> properties = EquivalentDownProperties((Property)
                             entity);
                        for (Property p : properties) {
                                if (p instanceof FunctionDec) {
                                        equiv.add(((FunctionDec) p).
                                            getFunctionResultEntity());
                                } else {
                                        equiv.add((Attribute) p);
                                }
                        }
                } else if (entity instanceof FormalArgument) {
                        Method md = ((FormalArgument) entity).getMethDec();
                        List<Property> properties = EquivalentDownProperties(md);
                        Integer index = md.getIndexFormalArg((FormalArgument) entity);
                        for (Property p : properties) {
                                if (p instanceof Method) {
                                        Method temp = (Method) p;
                                        List<FormalArgument> arguments = temp.
                                            getFormalArgument();
                                        if (arguments.size() > index) {
```

```
                                                    equiv.add(arguments.toSequence()[index
                                                        ]);
                                                }
                                            }
                                        }
                                    }
                                }
        return equiv;
}
```

D.4. GDD

Descripción: devuelve el conjunto de entidades genérico dependientes directas de la entidad de entrada. Una entidad genérico dependiente directa tendrá como nuevo tipo el definido por el parámetro formal añadido en la refactorización. Se calcula como resultado de la ejecución del algoritmo definido en **GenericDependant** (ver Código D.3.6).

Código D.4.33: Función GDD

```
namespace Moon::Function;

activity GDD(in entity : Entity) : List<Entity> {
        List<Entity> gdi = new List<Entity>();
        return GenericDependant(entity, gdi);
}
```

D.5. GDI

Descripción: devuelve el conjunto de entidades genérico dependientes indirectas de la entidad de entrada. Una entidad genérico dependiente indirecta tendrá como nuevo tipo, un tipo paramétrico no completo, que depende del nuevo parámetro formal añadido en la refactorización. Al igual que con la función **GDD**, se calcula a partir del algoritmo definido en **GenericDependant** (ver Código D.3.6).

Código D.5.34: Función GDI

```
namespace Moon::Function;

activity GDI(in entity : Entity) : List<Entity> {
        List<Entity> gdi = new List<Entity>();
        GenericDependant(entity, gdi);
        return gdi;
}
```

D.6. ExpressionsAssignedToEntity

Descripción: consulta el conjunto de expresiones asignadas a una cierta entidad en el código.

Código D.6.35: Función `ExpressionsAssignedToEntity`

```
namespace Moon::Function;

public import Moon::ExpressionsAssignedToEntityUtilities::VisitInstructionsInBodyMethod
    ;

activity ExpressionsAssignedToEntity(in entity : Entity) : List<Expr> {
        List<Expr> result = new List<Expr>();
        if (entity.getClassDef() != null) { // has class definition
                if (entity instanceof Attribute) {
                        Attribute ad = ((Attribute) entity);
                        List<Method> list = ad.getClassDef().getMethods();
                        for (Method md : list) {
                                List<Instr> body = md.getFlattenedInstructions();
                                VisitInstructionsInBodyMethod(body, entity, result);
                        }
                } else if (entity instanceof Local) {
                        Method md = ((Local) entity).getMethDec();
                        List<Instr> body = md.getFlattenedInstructions();
                        VisitInstructionsInBodyMethod(body, entity, result);
                } else if (entity instanceof FormalArgument) {
                        Method md = ((FormalArgument) entity).getMethDec();
                        List<Instr> body = md.getFlattenedInstructions();
                        VisitInstructionsInBodyMethod(body, entity, result);
                }
        }
        return result;
}
```

Dicha función se apoya en la función auxiliar **`VisitInstructionsInBodyMethod`** definida tal y como se muestra en el Código D.6.36.

Código D.6.36: Función `VisitInstructionsInBodyMethod`

```
namespace Moon::ExpressionsAssignedToEntityUtilities;

activity VisitInstructionsInBodyMethod(in body : List<Instr>, in entity : Entity, inout
     listExpr : List<Expr>) {
        for (Instr instr : body) {
                if (instr instanceof CompoundInstr) {
                        VisitInstructionsInBodyMethod(((CompoundInstr)instr).
                            getInstructions(), entity, listExpr);
                } else {
                        // visit instr leaf
                        if (instr instanceof AssignmentInstr) {
                                Expr exprLeft = ((AssignmentInstr) instr).getLeftSide()
                                    ;
                                if (exprLeft instanceof CallExpr
                                        && ((CallExpr) exprLeft).getTarget() ==  entity
                                            ) {
                                        Expr exprRight = ((AssignmentInstr) instr).
                                            getRightSide();
                                        if (!listExpr.includes(exprRight)) {
                                                listExpr.add(exprRight);
                                        }
                                }
                        }
                }
        }
}
```

D.7. CallExprUsedByEntity

Descripción: consulta el conjunto de expresiones utilizadas a través de una cierta entidad en el código.

Código D.7.37: Función `CallExprUsedByEntity`

```
namespace Moon::Function;

public import Moon::CallExprUsedByEntityUtilities::VisitInstructionsInBodyMethod;

activity CallExprUsedByEntity(in entity : Entity) : List<CallExpr> {
        List<CallExpr> result = new List<CallExpr>();
        if (entity.getClassDef() != null) { // has class definition
                if (entity instanceof Attribute) {
                        Attribute ad = ((Attribute) entity);
                        List<Method> list = ad.getClassDef().getMethods();
                        for (Method md : list) {
                                List<Instr> body = md.getFlattenedInstructions();
                                VisitInstructionsInBodyMethod(body, entity, result);
                        }
                } else if (entity instanceof Local) {
                        Method md = ((Local) entity).getMethDec();
                        List<Instr> body = md.getFlattenedInstructions();
                        VisitInstructionsInBodyMethod(body, entity, result);
                } else if (entity instanceof FormalArgument) {
                        Method md = ((FormalArgument) entity).getMethDec();
                        List<Instr> body = md.getFlattenedInstructions();
                        VisitInstructionsInBodyMethod(body, entity, result);
                }
        }
        return result;
}
```

Dicha función se apoya en la función auxiliar **VisitInstructionsInBodyMethod** definida tal y como se muestra en el Código D.7.38, junto con la función auxiliar **VisitCallExpr**, descrita en el Código D.7.39.

Código D.7.38: Función `VisitInstructionsInBodyMethod`

```
namespace Moon::CallExprUsedByEntityUtilities;

activity VisitInstructionsInBodyMethod(in body : List<Instr>, in entity : Entity, inout
     listExpr : List<CallExpr>) {
        for (Instr instr : body) {
                if (instr instanceof CompoundInstr) {
                        VisitInstructionsInBodyMethod(((CompoundInstr)instr).
                            getInstructions(), entity, listExpr);
                } else {
                        // visit instr leaf
                        if (instr instanceof AssignmentInstr) {
                                AssignmentInstr assignmentInstr = (AssignmentInstr)
                                    instr;
                                if (assignmentInstr.getLeftSide() instanceof CallExpr){
                                     VisitCallExpr((CallExpr) assignmentInstr.
                                         getLeftSide(), entity, listExpr);
                                }
                                if (assignmentInstr.getRightSide() instanceof CallExpr)
                                    {
```

```
                                        VisitCallExpr((CallExpr) assignmentInstr.
                                            getRightSide(), entity, listExpr);
                                }
                        }
                        else if (instr instanceof CallInstr) {
                                CallInstr callInstr = (CallInstr) instr;
                                // visit arguments
                                for (Expr expr : callInstr.getRealArguments()){
                                        if (expr instanceof CallExpr) {
                                                VisitCallExpr((CallExpr) expr, entity,
                                                    listExpr);
                                        }
                                }
                                CallExpr leftSide = (CallExpr) callInstr.getLeftSide();
                                VisitCallExpr(leftSide, entity, listExpr);
                        }
                }
        }
}
```

Código D.7.39: Función `VisitCallExpr`

```
namespace Moon::CallExprUsedByEntityUtilities;

activity VisitCallExpr(in callExpr : CallExpr, in entity : Entity, inout listExpr :
    List<CallExpr>) {
        CallExpr auxExpr = callExpr;
        while (auxExpr != null)  {
                if (auxExpr.getTarget() == entity &&
                        auxExpr.getRightSide() != null) {
                        CallExpr aux = (CallExpr) auxExpr.getRightSide();
                        while(aux.getRightSide()!=null && aux.getRightSide() instanceof
                             CallExpr) {
                                aux= (CallExpr) aux.getRightSide();
                        }
                        if (!listExpr.includes(aux)) {
                                listExpr.add((CallExpr)aux);
                        }
                }
                // visit arguments
                for (Expr expr : auxExpr.getRealArguments()){
                        if (expr instanceof CallExpr) {
                                VisitCallExpr((CallExpr) expr, entity, listExpr);
                        }
                }
                auxExpr = (CallExpr) auxExpr.getLeftSide();
        }
}
```

D.8. CallExprUsingEntity

Descripción: consulta el conjunto de expresiones que utilizan a una cierta entidad en el código.

Código D.8.40: Función `CallExprUsingEntity`

```
namespace Moon::Function;
```

```
public import Moon::CallExprUsingEntityUtilities::VisitInstructionsInBodyMethod;

activity CallExprUsingEntity(in entity : Entity) : List<CallExpr> {
        List<CallExpr> result = new List<CallExpr>();
        if (entity.getClassDef() != null) { // has class definition
                if (entity instanceof Attribute) {
                        Attribute ad = ((Attribute) entity);
                        if (ad.hasInitializer()
                                        && !(ad.getInitializer() instanceof
                                            ManifestConstant)) {
                                if (!result.includes((CallExpr)ad.
                                    getInitializer())) {
                                        result.add((CallExpr)ad.getInitializer
                                            ());
                                }
                        }
                        List<Method> list = ad.getClassDef().getMethods();
                        for (Method md : list) {
                                List<Instr> body = md.getFlattenedInstructions();
                                VisitInstructionsInBodyMethod(body, entity, result);
                        }
                } else if (entity instanceof Local) {
                        Method md = ((Local) entity).getMethDec();
                        List<Instr> body = md.getFlattenedInstructions();
                        VisitInstructionsInBodyMethod(body, entity, result);
                } else if (entity instanceof FormalArgument) {
                        Method md = ((FormalArgument) entity).getMethDec();
                        List<Instr> body = md.getFlattenedInstructions();
                        VisitInstructionsInBodyMethod(body, entity, result);
                }
        }
        return result;
}
```

Dicha función se apoya en la función auxiliar **VisitInstructionsInBodyMethod** definida tal y como se muestra en el Código D.8.41, junto con la función auxiliar **VisitCallExpr**, descrita en el Código D.8.42.

Código D.8.41: Función `VisitInstructionsInBodyMethod`

```
namespace Moon::CallExprUsingEntityUtilities;

activity VisitInstructionsInBodyMethod(in body : List<Instr>, in entity : Entity, inout
     listExpr : List<CallExpr>) {
        for (Instr instr : body) {
                if (instr instanceof CompoundInstr) {
                        VisitInstructionsInBodyMethod(((CompoundInstr)instr).
                            getInstructions(), entity, listExpr);
                } else {
                        // visit instr leaf
                        if (instr instanceof AssignmentInstr) {
                                AssignmentInstr assignmentInstr = (AssignmentInstr)
                                    instr;
                                if (assignmentInstr.getLeftSide() instanceof CallExpr){
                                        VisitCallExpr((CallExpr) assignmentInstr.
                                            getLeftSide(), entity, listExpr);
                                }
                                if (assignmentInstr.getRightSide() instanceof CallExpr)
                                    {
```

```
                                        VisitCallExpr((CallExpr) assignmentInstr.
                                            getRightSide(), entity, listExpr);
                                }
                        }
                }
        }
}
```

Código D.8.42: Función `VisitCallExpr`

```
namespace Moon::CallExprUsingEntityUtilities;

activity VisitCallExpr(in callExpr : CallExpr, in entity : Entity, inout listExpr :
    List<CallExpr>) {
        CallExpr auxExpr = callExpr;
        while (auxExpr != null)  {
                if (auxExpr.getTarget() == entity) {
                        if (!listExpr.includes(auxExpr)) {
                                listExpr.add((CallExpr)auxExpr);
                        }
                }
                // visit arguments
                for (Expr expr : auxExpr.getRealArguments()){
                        if (expr instanceof CallExpr) {
                                VisitCallExpr((CallExpr) expr, entity, listExpr);
                        }
                }
                auxExpr = (CallExpr) auxExpr.getLeftSide();
        }
}
```

D.9. AddFormalParameterInCandidateClassesAction

Descripción: añade en las clases candidatas a parametrizar el nuevo parámetro formal resultado de la parametrización, cambiando los tipos en las cláusulas de herencia si fuera necesario.

Código D.9.43: Acción `AddFormalParameterInCandidateClassesAction`

```
namespace Moon::Action;

activity AddFormalParameterInCandidateClassesAction(in candidates : List<ClassDef>, in
    name : Name, in type : Type) {
        for (ClassDef cd : candidates) {
                FormalPar fp = Model.getModel().getMoonFactory().createFormalPar(name);
                // with boundS add bound type, with where clauses add properties
                ((BoundS) fp).addBound(type);
                cd.add(fp); // add formal par
                ChangeInheritanceClausesAction(cd, type, candidates, fp);
        }
}
```

Dicha acción se apoya en la acción auxiliar **ChangeInheritanceClausesAction** definida tal y como se muestra en el Código D.9.44 y en la acción auxiliar **AddTypeInInheritanceClauseAction** definida tal y como se muestra en el Código D.9.45. La primera

acción selecciona el tipo sobre el que se realiza el cambio en la cláusula de herencia, utilizando la segunda acción para cambiar el tipo de la cláusula en caso de que fuera necesario.

Código D.9.44: Acción `ChangeInheritanceClausesAction`

```
namespace Moon::Action;

activity ChangeInheritanceClausesAction(in classDef : ClassDef, in entityType : Type,
    in candidates : List<ClassDef>, in fp : FormalPar) {
        for (ClassDef cd : classDef.getDirectDescendants()) {
                for (InheritanceClause ic : cd.getInheritanceClause()) {
                        if (candidates.includes(cd)) { // is a candidate class in gdi
                                AddTypeInInheritanceClauseAction(classDef, fp, ic); //
                                    add formal parameter
                        }
                        else if (ic.getType().getClassDef() == classDef) {
                                AddTypeInInheritanceClauseAction(classDef, entityType,
                                    ic); // add type
                        }
                }
        }
}
```

Código D.9.45: Acción `AddTypeInInheritanceClauseAction`

```
namespace Moon::Action;

activity AddTypeInInheritanceClauseAction(in classDef : ClassDef, in type : Type, in ic
     : InheritanceClause) {
        List<Type> realParameters = new List<Type>();
        // if it is parametric type, add the previous real parameters before...
        if (ic.getType() instanceof ClassType && ((ClassType) ic.getType()).
            isParametric()) {
                List<Type> real = ((ClassType) ic.getType()).getRealParameters();
                realParameters.addAll(real);
        }
        // add new real parameter
        realParameters.add(type);
        Type newAncestorType = Model.getModel().getMoonFactory().createParametricType(
            classDef.getName(), classDef,      realParameters);
        ic.set(newAncestorType); // change the inheritance clause
}
```

D.10. ChangeGDDEntitiesAction

Descripción: cambia el tipo de la entidades genérico dependientes directas por el parámetro formal añadido.

Código D.10.46: Acción `ChangeGDDEntitiesAction`

```
namespace Moon::Action;

public import Moon::Function::GDD;

activity ChangeGDDEntitiesAction(in targetEntity : Entity, in formalParName : Name) {

        // change the GDD entities...
```

```
        List<Entity> gddEntities = GDD(targetEntity);

        for (Entity entity : gddEntities) {
                if (!(entity instanceof Self)) {

                        // if the entity has a container class...
                        if (entity.getClassDef() != null) {
                                // gets the formal parameter
                                fp = entity.getClassDef().getFormalPar(formalParName);
                                if (fp != null) {

                                        if (entity instanceof Result) {
                                                // if it is a result, change the return
                                                     type of the
                                                // function
                                                fd = (FunctionDec) ((Result) entity)
                                                                .getFunctionDec();
                                                Type newFp = ((Result) entity).getType
                                                    ().getTransformedType(fp);
                                                fd.setFunctionResultEntity(newFp);
                                        }
                                        else {
                                                fd = null; // Alf constraints
                                                // in other case, usual solution
                                                Type newFp = entity.getType().
                                                    getTransformedType(fp);
                                                entity.setType(newFp);
                                        }
                                }
                        }
                }
        }
}
```

D.11. ChangeGDIEntitiesAction

Descripción: cambia el tipo de la entidades genérico dependientes indirectas añadiendo el parámetro formal en la declaración de tipo.

Código D.11.47: Acción ChangeGDIEntitiesAction

```
namespace Moon::Action;

public import Moon::Function::GDI;
public import Moon::Function::CreateNewParametricType;

activity ChangeGDIEntitiesAction(in historyClasses : List<ClassDef>, in historyEntity :
     List<Entity>, in candidates : List<ClassDef>, in entity : Entity, in formalParName
    : Name) {

        // Change the gdi entities...
        List<Entity> gdiEntities = GDI(entity);
        for (Entity entityAux : gdiEntities) {

                if (!(entityAux instanceof Self)) {
                        FormalPar fp = Model.getModel().getMoonFactory().
                            createNullFormalPar();
                        if (entityAux instanceof Attribute) {
```

```
                        fp = ((Attribute) entityAux).getClassDef().getFormalPar
                            (formalParName);
                    } else if (entityAux.getClassDef() != null) {
                        // gets the formal parameter
                        fp = entityAux.getClassDef().getFormalPar(formalParName
                            );
                    }
                    Type oldType = entity.getType();
                    if (fp != null) {
                        if (oldType.getClassDef().isGeneric()) {
                            ClassType newClassType =
                                CreateNewParametricType(fp,
                                            oldType);
                            entity.setType(newClassType);
                            historyEntity.add(entityAux);
                            // for each entity in GDI change the clients...
                            FormalPar fp2 = entity.getClassDef().
                                getFormalPar(formalParName);
                            ChangeClientsOfEntityAction(entity, candidates,
                                 historyClasses, historyEntity, fp2);
                        }
                    }
                    else {
                        if (entityAux.getType().getClassDef().isGeneric()
                                && !historyEntity.includes(entityAux))
                                    {

                            ClassDef cd = entityAux.getType().getClassDef()
                                ;
                            List<Type> realParameters = new List<Type>();
                            if (oldType instanceof ClassType && ((ClassType
                                ) oldType).isParametric()) {
                                real = ((ClassType) oldType).
                                    getRealParameters();
                                realParameters.addAll(real);
                            }
                            FormalPar fpAux = Model.getModel().
                                getMoonFactory().createFormalPar(
                                formalParName);
                            // with boundS add bound type, with where
                                clauses add properties
                            ((BoundS) fpAux).addBound(oldType);
                            cd.add(fpAux); // add formal par

                            ClassType newClassType =
                                CreateNewParametricType(entity.getType(),
                                entityAux.getType());
                            entityAux.setType(newClassType);
                            historyEntity.add(entityAux);
                        } // if
                    } //  else
                } // if
            } // for
}
```

Dicha acción se apoya en la acción auxiliar **ChangeClientsOfEntityAction** definida tal y como se muestra en el Código D.11.48 y en la función auxiliar **CreateNewParametricType** definida tal y como se muestra en el Código D.11.49. La primera acción se cambia el tipo de entidades clientes, creando nuevos tipos paramétricos a través del tipo calculado con la función.

Código D.11.48: Acción `ChangeClientsOfEntityAction`

```
namespace Moon::Action;

public import Moon::Function::Entities;
public import Moon::Function::CreateNewParametricType;

activity ChangeClientsOfEntityAction(in entity : Entity, in candidates: List<ClassDef>,
     in historyClasses : List<ClassDef>, in historyEntity : List<Entity>, in formalPar
    : FormalPar) {
        // get direct clientso of the class of the initial entity
        List<ClassDef> directClients = entity.getClassDef().getDirectClient();
        for (ClassDef dClient : directClients) {
                if (!candidates.includes(dClient)
                                && !historyClasses.includes(dClient)) {
                        // Change entities in clients...
                        historyClasses.add(dClient);
                        List<Entity> entities = Entities(dClient);
                        for (Entity clientEntity : entities) {
                                if (!historyEntity.includes(clientEntity)) {
                                        if (clientEntity.getType() instanceof ClassType
                                                        && clientEntity.getType().
                                                            getClassDef()
                                                                        == entity.
                                                                            getClassDef
                                                                            ()) {
                                                Type oldType = entity.getClassDef().
                                                    getClassType();
                                                Type newType = CreateNewParametricType(
                                                    entity.getType(), oldType);
                                                clientEntity.setType(newType);
                                                historyEntity.add(clientEntity);
                                        }
                                }
                        } // for
                } // if
        } // for
}
```

Código D.11.49: Función `CreateNewParametricType`

```
namespace Moon::Function;

activity CreateNewParametricType(in oldType : Type,  in type : Type) : ClassType {
        ClassDef cd = oldType.getClassDef();
        List<Type> list = new List<Type>();
        if (oldType instanceof ClassType
                                && ((ClassType) oldType).isParametric()) {
                        List<Type> real = ((ClassType) oldType).getRealParameters();
                        list.addAll(real);
                }
        list.add(type);
        ClassType newClassType = (ClassType) (Model.getModel().getMoonFactory().
            createParametricType(cd.getName(), cd, list));
        return newClassType;
}
```

D.12. ChangeClientsOfGDIEntitiesAction

Descripción: cambia el tipo en los clientes de entidades genérico dependientes indirectas.

Código D.12.50: Acción `ChangeClientsOfGDIEntitiesAction`

```
namespace Moon::Action;

public import Moon::Function::CreateNewParametricType;

activity ChangeClientsOfGDIEntitiesAction(in historyEntity : List<Entity>, in
    candidates : List<ClassDef>, in oldTargetEntityType : Type){
        // change clients of GDI entities in client classes
        // for each candidate class add the formal parameter...
        // for example D to D<C> in a class B without formal parameter
        for (ClassDef cd : candidates) {
                List<ClassDef> listDirectClients = cd.getDirectClient();
                for (ClassDef directClient : listDirectClients) {
                        // remove the current class from the list
                        if (!(directClient == cd)
                                        && directClient.isSourceAvailable()) {
                                List<Entity> list = directClient.getEntities();
                                for (Entity entity : list) {
                                        if (entity.getType().getClassDef() != null
                                                        && entity.getType().getClassDef
                                                            () == cd
                                                        && !historyEntity.includes(
                                                            entity)) {
                                                Type newClassType =
                                                    CreateNewParametricType(
                                                                oldTargetEntityType,
                                                                    entity.getType());
                                                entity.setType(newClassType);
                                                historyEntity.add(entity);
                                        }
                                }
                        }
                }
        }
}
```

Utiliza la función **`CreateNewParametricType`** (ver Código D.11.49) para calcular el nuevo tipo.

APÉNDICE E

PUBLICACIONES

Esta tesis surge para resolver parcialmente 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 se creó un grupo de trabajo de cuatro miembros: tres doctorandos, entre los que se incluye el autor, y como directora del grupo, la Dra. Yania Crespo. En dicho ámbito se establecen tres temas de trabajo: detección de defectos, corrección y definición de refactorizaciones, siendo este último el objeto de esta tesis.

Los capítulos de la tesis han sido tratados en alguna publicación de taller o congreso. Estas publicaciones han servido por una lado, para validar las propuestas, y por otro lado, mejorar las propuestas con los comentarios de sus revisores.

A continuación se indican los congresos y talleres 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.

E.1. Congresos y talleres

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:

- ICSR (*International Conference on Software Reuse*). Conferencia en reutilización del software donde se participó en su edición del 2004 en la sesión de *posters*, mostrando la propuesta inicial del trabajo.

- CAISE Doctoral Consortium (*Conference on Advanced Information Systems Engineering*). Dentro la conferencia internacional CAISE se desarrolla un taller para aquellos doctorandos que inician su trabajo de investigación, recibiendo recomendaciones y guías de los investigadores más expertos. En dicho taller, en el año 2005, se presentó el trabajo inicial, a partir de cuál se establecieron los fundamentos del presente trabajo.
- 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 dos ediciones del taller: novena (2005) y décima (2006)), en particular con el uso del metamodelo MOON como solución de soporte para la obtención de métricas, y posterior utilización en la detección de defectos. Este taller ha estado integrado dentro del congreso de relevancia internacional ECOOP (European Conference on Object-Oriented Programming).
- WOOR *(Workshop on Object-Oriented Reengineering*), es un taller especializado en reingeniería y evolución del software. Se ha participado con publicaciones 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).
- WRT (*Workshop on Refactoring Tools*), es un taller especializado en la presentación de herramientas de refactorización, integrado dentro de la conferencia ECOOP (en sucesivas ediciones también en OOPSLA o ICSE). Se participó en la primera edición presentando la arquitectura propuesta del presente trabajo, centrado en las virtudes de la reutilización para la definición y construcción de refactorizaciones.

Los congresos en los que se ha participado están centrados el contexto de la Ingeniería del Software y han servidor 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. Desde 2003 se ha participado en ocho ediciones de este congreso, con presentación de artículos y herramientas.
- PROLE (Jornadas de Programación y Lenguajes) es un congreso de perfil genérico sobre lenguajes de programación. Se participó de manera puntual en el 2003 con un artículo, presentando una primera versión del catálogo de refactorizaciones en genericidad, en particular las vinculadas a la especialización.
- ICSOFT (*International Conference on Software and Data Technologies*), el congreso está organizado en áreas relacionadas con nuevas tendencias en la informática e ingeniería del software. Se ha participado en tres del las ediciones del mismo, dentro del área de Ingeniería del Software y mantenimiento y evolución del software.
- CSMR (*Conference on Software Maintenance and Reengineering*), conferencia centrada en las labores de mantenimiento y reingeniería. Se participó en la edición del año 2010, presentando una aplicación práctica de la solución propuesta a la refactorización **Extract Method**.

Una visión integradora de los objetivos de investigación generales a GIRO fue publicada como capítulo del libro titulado *Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices* [Crespo et al., 2006a].

También se ha participado en alguna conferencia de enfoque de innovación docente, como JENUI (Jornadas de Enseñanza Universitaria de la Informática), pero con alguna publicación centrada en el estudio de las refactorizaciones y su caracterización, cuyos resultados finales han tenido algún reflejo en el presente trabajo.

En paralelo, y dentro de las actividades realizadas en el grupo GIRO, con relación a las otras dos líneas de investigación marcada, se ha participado también en algunos trabajos sobre métricas y detección de defectos, como en la *Revista de Procesos y Métricas*.

E.2. Publicaciones y relación con los capítulos

A continuación se detallan las publicaciones realizadas y los capítulos relacionados, en orden cronológico. Las publicaciones están disponibles en el repositorio del grupo de investigación GIRO[1]. En la Tabla E.1 se recoge un cuadro resumen de las referencias cruzadas entre las distintas publicaciones realizadas y los capítulos, como resumen final.

En relación con el Capítulo 2, como base del trabajo de investigación [Marticorena and Crespo, 2003] se realizó un primer estudio del estado del arte en refactorización, particularmente sobre cuestiones de especialización de código. Es a partir de este trabajo de investigación del cuál se establecieron las hipótesis iniciales, objetivos, metodología y beneficios esperados de esta tesis.

A partir de dichos resultados, en [López et al., 2003], se recogió una primera propuesta de la definición de la plantilla de refactorizaciones con independencia del lenguaje, y una primera aproximación al uso de *frameworks* como solución, en relación al Capítulo 3, Capítulo 4 y Capítulo 6.

En [Marticorena et al., 2003] se profundizó sobre el primer desarrollo de un catálogo de refactorizaciones en especialización, en cuanto a genericidad, en relación al Capítulo 4 y Capítulo 5.

Posteriormente, en [Crespo et al., 2004] se extendió el anterior trabajo, en relación al Capítulo 4 y Capítulo 6, más concretamente en relación a la ejecución de las refactorizaciones, dejando una sólida base para su ejecución. En [López et al., 2004] se presentó un póster con la visión global, en base a los anteriores trabajos, de lo que son los fundamentos para el resto de trabajos desarrollados, haciendo foco principal en el Capítulo 6.

Como resultado del análisis de las refactorizaciones, en relación con el Capítulo 3 se presentó en [Marticorena et al., 2004] un estudio de las refactorizaciones y sus características para facilitar su clasificación, en relación con el Capítulo 4.

[1] Pueden consultarse en `http://www.giro.infor.uva.es/Publications/index.php?idAuthor=12`

En [Crespo et al., 2005a] y [Crespo et al., 2005b], en relación con el Capítulo 3 y el Capítulo 6 se estudió cómo consultar la información almacenada con el fin de sugerir oportunidades de refactorización, a través de métricas y heurísticas. Se prosiguió con el uso de *frameworks* para la obtención de métricas, a través del diseño de un *framework* que permita obtener esta información sobre el lenguaje minimal MOON, con independencia del lenguaje.

En esta línea se constató la posibilidad de obtener métricas independientes del lenguaje sobre plataformas como JAVA y .NET. Siguiendo en esta línea, en [Marticorena et al., 2005] se presentó el uso de métricas y heurísticas independientes del lenguaje (consultas), en relación con el Capítulo 3, para sugerir la detección de defectos de código y poder aplicar el conjunto de refactorizaciones.

En [Marticorena, 2005] en relación con el Capítulo 3 y Capítulo 4 se presentó la arquitectura propuesta, refinando también lo expuesto en [López et al., 2004] en relación al Capítulo 6, detectando defectos en trabajos anteriores, corrigiendo y mejorando la propuesta.

En [Crespo et al., 2006a] se participó mediante el desarrollo de un capítulo del libro, con una revisión completa del Capítulo 2 hasta el Capítulo 6.

En [Crespo et al., 2006b] y [Marticorena et al., 2006] se prosiguió con la clasificación y mecanismos para la detección de oportunidades de refactorización en relación con el Capítulo 6.

En [López et al., 2006a] se planteó un modelo de caracterización con el fin de clasificar y asistir a la construcción de refactorizaciones en base a lo planteado en el Capítulo 3 y Capítulo 4.

En [López et al., 2006b] 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 (como precedente al uso de ALF), en relación con el Capítulo 3 y Capítulo 6. Dado el nivel de complejidad de UML y el ámbito de aplicación de las refactorizaciones objeto del estudio, se mostraron las ventajas del uso de una representación minimal como MOON.

En [Marticorena et al., 2007a] 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, bajo un proceso claramente marcado. El trabajo se encuadra dentro del Capítulo 4.

En [Marticorena et al., 2007b] se presentó una evolución del motor de refactorizaciones, y más concretamente sobre las instanciaciones concretas del mismo, las refactorizaciones y sus elementos, para permitir una construcción más dinámica en base al proceso planteado en [Marticorena et al., 2007a]. Se encuadra dentro del Capítulo 6 y Capítulo 7 concretamente en el soporte basado en *frameworks* y se apunta la evolución lógica del uso de *frameworks* de caja blanca hacia *frameworks* de caja gris. La herramienta construida apunta las opciones de reutilizar elementos ya construidos, facilitando la progresiva construcción de nuevas refactorizaciones.

Tabla E.1: Referencias cruzadas entre publicaciones y módulos

Publicación \ Capítulo	Congreso/Taller	C2	C3	C4	C5	C6	C7
[Marticorena and Crespo, 2003]	Technical Report	√	-	-	-	-	-
[López et al., 2003]	JISBD	-	√	√	-	√	-
[Marticorena et al., 2003]	PROLE	-	-	√	√	-	-
[Crespo et al., 2004]	JISBD	-	-	√	-	√	-
[López et al., 2004]	ICSR	-	-	-	-	√	-
[Marticorena et al., 2004]	JENUI	-	√	√	-	-	-
[Crespo et al., 2005a]	JISBD	-	√	-	-	√	-
[Crespo et al., 2005b]	QAOOSE	-	√	-	-	√	-
[Marticorena et al., 2005]	WOOR	-	√	-	-	-	-
[Marticorena, 2005]	CAISE	-	√	√	-	√	-
[Crespo et al., 2006a]	Capítulo de libro	√	√	√	√	√	-
[Crespo et al., 2006b]	QAOOSE	-	-	-	-	√	-
[Marticorena et al., 2006]	WOOR	-	-	-	-	√	-
[López et al., 2006a]	JISBD	-	√	√	-	-	-
[López et al., 2006b]	ICSOFT	-	√	-	-	√	-
[Marticorena et al., 2007a]	JISBD	-	-	√	-	-	-
[Marticorena et al., 2007b]	WRT	-	-	-	-	√	√
[Marticorena and Crespo, 2008]	ICSOFT	-	√	√	√	√	√
[Marticorena et al., 2008]	JISBD	-	-	-	-	√	√
[Marticorena et al., 2010a]	CSMR	-	-	-	-	√	√
[Marticorena et al., 2010b]	Rev.Proc.y Mét.	-	-	-	-	√	-
[Marticorena et al., 2011b]	JISBD	-	-	-	√	√	√
[Marticorena et al., 2011a]	JISBD	-	-	-	√	√	√
[Marticorena et al., 2011c]	ICSOFT	-	-	√	-	-	-
[Marticorena and Crespo, 2012]	JISBD	-	-	√	√	-	-

Sobre la base de dicho trabajo, en [Marticorena and Crespo, 2008] se profundizó sobre la oportunidad de trabajar desde una solución declarativa, desde el punto de vista de la reutilización, abarcando desde el Capítulo 3 al Capítulo 7 y las facilidades que esto aporta para el desarrollo de refactorizaciones. Aunque la herramienta en desarrollo entra en la categoría inicial de prototipo, se apuntan cuestiones sobre su migración como *plugin* en entornos más complejos como Eclipse.

En [Marticorena et al., 2008] se presenta un estudio de la aplicabilidad de las refactorizaciones y de la solución propuesta al problema de la migración del software entre distintas versiones de bibliotecas, en concreto tomando como caso de estudio Java y JUnit. La propuesta se encuadra en el Capítulo 6 y Capítulo 7, y se hace alguna reflexión adicional en el Capítulo 8.

Vinculado a los Capítulo 6 y Capítulo 7 se presenta un estudio e implementación de una refactorización bien conocida como **Extract Method** en [Marticorena et al., 2010a]

teniendo en cuenta cuestiones relativas a la genericidad. Se revisa el actual estado del arte y cómo la propuesta realizada resuelve ciertos problemas en cuanto a la inclusión de la genericidad en refactorizaciones, actualmente soportadas por otras herramientas.

En [Marticorena et al., 2010b] se prosigue con la clasificación y mecanismos para la detección de oportunidades de refactorización en relación con el Capítulo 6.

En [Marticorena et al., 2011b] se presenta el estudio de las refactorizaciones como solución para el rejuvenecimiento de código. Se realizan estudios sobre la parametrización de clases así como la modificación de anotaciones de redefinición, mostrando la aplicabilidad de la solución, en relación con el Capítulo 5, Capítulo 6 y Capítulo 7. En dicho congreso, se presentó también el prototipo desarrollado en [Marticorena et al., 2011a], en relación con los mismos capítulos.

Prosiguiendo con el estudio en mayor profundidad de la caracterización de refactorizaciones, se presenta en [Marticorena et al., 2011c] una revisión de la caracterización y su aplicación al catálogo completo de [Fowler, 1999]. Los resultados se recogen en el Capítulo 4.

Finalmente, en [Marticorena and Crespo, 2012] se presenta el uso del lenguaje ALF como lenguaje de especificación de refactorizaciones, resolviendo algunos de los problemas detectados habitualmente en la definición de las refactorizaciones, tal y como se explica en el Capítulo 4 y Capítulo 5.

BIBLIOGRAFÍA

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[Aho et al., 1990] Aho, A. V., Sethi, R., and Ullman, J. D. (1990). *Compiladores. Principios, técnicas y herramientas*. Pearson Addison-Wesley.

[Alanen et al., 2003] Alanen, M., Porres, I., Centre, T., and Science, C. (2003). A relation between context-free grammars and meta object facility metamodels. Technical report, TUCS Turku Center for Computer Science. Abo Akademi University.

[Ambler and Sadalage, 2006] Ambler, S. W. and Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional.

[Baker, 1972] Baker, F. T. (1972). Chief programmer team management of production programming. *IBM Syst. J.*, 11(1):56–73.

[Beck, 1999] Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1st edition.

[Beck and Andres, 2004] Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition.

[Beck and Gamma, 1998] Beck, K. and Gamma, E. (1998). Test Infected: Programmers Love Writing Tests. *Java Report*, 3(7):37–50.

[Becker, 2011] Becker, P. (2011). Working Draft, Standard for Programming Language C++. Technical Report N3242=11-0012, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++.

[Beckert et al., 2002] Beckert, B., Keller, U., and Schmitt, P. H. (2002). Translating the Object Constraint Language into First-order Predicate Logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, pages 113–123.

[Bennett and Rajlich, 2000] Bennett, K. H. and Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA. ACM.

[Bishop, 2008] Bishop, J. M. (2008). *C# 3.0 Design Patterns*. O'Reilly, 1st edition.

[Bär, 1999] Bär, H. (1999). FAMIX C++ Language Plug-in 1.0. Technical report, University of Berne. Available at http://scg.unibe.ch/archive/famoos/FAMIX/Plugins/C++Plugin1.0.ps.gz.

[Bracha et al., 2001] Bracha, G., Cohen, N., Kemper, C., Marx, S., Odersky, M., Panitz, S.-E., Stoutamire, D., and andPhilip Wadler, K. T. (2001). Adding Generics to the Java Programming Language: Participant Draft Specification.

[Bracha et al., 1998a] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998a). GJ: Extending the Java Programming Language with Type Parameters. Manuscript. Available at http://lampw3.epfl.ch/gj/Documents/gj-tutorial.pdf.

[Bracha et al., 1998b] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998b). GJ: Specification. Manuscript. Available at http://lampw3.epfl.ch/gj/Documents/gj-specification.pdf.

[Bracha et al., 1998c] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998c). Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Chambers, C., editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC.

[Bravo, 2003] Bravo, F. M. (2003). A Logic Meta-Programming Framework for Supporting the Refactoring Process. Master's thesis, Vrije Universiteit Brussel, Belgium and Ecole Des and Mines, Nantes.

[Bright, 2004] Bright, W. (2004). The D Programming Language. Available at http://www.digitalmars.com/d/sdwest/index.html.

[Brooks, 1995] Brooks, F. P. (1995). *The mythical man-month (20th Anniversary Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Buckley et al., 2005] Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. (2005). Towards a taxonomy of software change. *Software Maintenance and Evolution: Research and Practice*, 17(5):309–332.

[Canning et al., 1989] Canning, P. S., Cook, W., Hill, W. L., Mitchell, J. C., and Olthoff, W. G. (1989). F-bounded quantification for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 273–280.

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). On Understanding types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522.

[Casais, 1990] Casais, E. (1990). Managing Class Evolution in Object-Oriented Systems. Technical report, Centre Universitaire d´Informatique, University of Geneve.

[Casais, 1992] Casais, E. (1992). An Incremental Class Reorganization Approach. In Madsen, O. L., editor, *Proc. European Conf. Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 114–132. Springer-Verlag.

[Casais, 1994] Casais, E. (1994). Automatic reorganization of object-oriented hierarchies: a case study. *Object Oriented Systems*, 1:95–115.

[CEA, 2013] CEA (2013). Open Source Tool for Graphical UML2 modelling. Available at http://www.papyrusuml.org/.

[Chikofsky and Cross, 1990] Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17.

[Connolly and Begg, 2009] Connolly, T. and Begg, C. (2009). *Database systems: a practical approach to design, implementation, and management.* Addison-Wesley, 5th edition.

[Cook et al., 2006] Cook, S., Harrison, R., Lehman, M. M., and Wernick, P. (2006). Evolution in software systems: foundations of the SPE classification scheme: Research Articles. *J. Softw. Maint. Evol.*, 18:1–35.

[Crespo, 2000] Crespo, Y. (2000). *Incremento del Potencial de Reutilización del Software mediante Refactorizaciones.* PhD thesis, Universidad de Valladolid. Available at http://giro.infor.uva.es/Publications/2000/Cre00.

[Crespo et al., 2006a] Crespo, Y., López, C., Manso, E., and Marticorena, R. (2006a). *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. Available at http://www.amazon.com/Object-Oriented-Design-Knowledge-Principles-Heuristics/dp/1591408962.

[Crespo et al., 2004] Crespo, Y., López, C., and Marticorena, R. (2004). Un Framework para la Reutilizacion de la Definicion de Refactorizaciones. In Juan Hernandez, H. P., editor, *IX Jornadas Ingenieria del Software y Bases de Datos (JISBD 2004), Malaga, Spain ISBN: 84-688-8983-0*, pages 499–506. Available at http://giro.infor.uva.es/Publications/2004/CLM04/.

[Crespo et al., 2005a] Crespo, Y., López, C., and Marticorena, R. (2005a). Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones. In *X Jornadas Ingenieria del Software y Bases de Datos (JISBD 2005), Granada, Spain ISBN: 84-9732-434-X*, pages 59–66. Available at http://giro.infor.uva.es/Publications/2005/MCL05.

[Crespo et al., 2006b] Crespo, Y., López, C., and Marticorena, R. (2006b). 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. Available at http://giro.infor.uva.es/Publications/2006/CLM06.

[Crespo et al., 2005b] Crespo, Y., López, C., Marticorena, R., and Manso, E. (2005b). Language Independent Metrics Support towards Refactoring Inference. In *9th ECOOP Workshop on QAOOSE 05, Quantitative Approaches in Object-Oriented Software Engineering, Glasgow, UK. ISBN: 2-89522-065-4*, pages 18–29. Available at http://giro.infor.uva.es/Publications/2005/CLMM05.

[Crespo and Marqués, 2001] Crespo, Y. and Marqués, J. M. (2001). Definición de un marco de trabajo para el análisis de refactorizaciones de software. In *VI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2001), Almagro (Ciudad Real). Spain, ISBN 84-699-6275-2*, pages 297–310, Oscar Díaz, Arantza Illarramendi, Mario Piattini. Available at http://giro.infor.uva.es/Publications/2001/CM01.

[Data Access Technologies Inc, 2012] Data Access Technologies Inc (2012). Action Language for UML (Alf) Parser. Available at http://lib.modeldriven.org/MDLibrary/trunk/Applications/Alf-Reference-Implementation/dist/.

[Daum, 2005] Daum, B. (2005). *Eclipse 3 para desarrolladores Java*. Anaya Multimedia, 1st edition.

[DedaSys, 2011] DedaSys (2011). LangPop.com programming Language Popularity. Available at http://www.langpop.com/.

[Dehnert and Stepanov, 1998] Dehnert, J. C. and Stepanov, A. A. (1998). Fundamentals of generic programming. In Jazayeri, M., Loos, R., and Musser, D. R., editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11. Springer.

[Demeyer et al., 1999] Demeyer, S., Tichelaar, S., and Steyaert, P. (1999). FAMIX 2.0 - the FAMOOS information exchange model. Technical report, Institute of Computer Science and Applied Mathematic. University of Bern.

[Developer Express Inc., 2013] Developer Express Inc. (2013). Refactor Pro for Visual Studio .NET. Available at http://www.devexpress.com/.

[Dig, 2007] Dig, D. (2007). WRT'07 1st Worskhop on Refactoring Tools. In *WRT'07 1st Worskhop on Refactoring Tools*, ISSN 1436-9915. In ECOOP'07, 21th European Conference Object-Oriented Programming, Berlin, Germany, Danny Dig and Michael Cebulla.

[Dig, 2008] Dig, D. (2008). WRT'07 2nd Worskhop on Refactoring Tools. In *WRT'08 2nd Worskhop on Refactoring Tools*. In OOPSLA'08, 23th International Conference on Object Oriented Programming, Systems, Languages and Applications, Nashville, Tennessee, USA, Danny Dig.

[Dig and Batory, 2011] Dig, D. and Batory, D. S. (2011). WRT'11 4th Worskhop on Refactoring Tools. In *WRT'11 4th Worskhop on Refactoring Tools*. In ICSE'11, 33rd International Conference on Software Engineering, Waikiki, Honolulu, Hawaii.

[Donovan et al., 2004] Donovan, A., Kiežun, A., Tschantz, M. S., and Ernst, M. D. (2004). Converting java programs to use generic libraries. *SIGPLAN Not.*, 39(10):15–34.

[Ducasse et al., 2000] Ducasse, S., Lanza, M., and Tichelaar, S. (2000). Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proc. Int'l Symp. Constructing Software Engineering Tools (CoSET)*, pages 24–30.

[Ducasse et al., 2001] Ducasse, S., Lanza, M., and Tichelaar, S. (2001). The Moose Reengineering Environment. *Smalltalk Chronicles*.

[Duffy, 2006] Duffy, J. (2006). *Professional .NET Framework 2.0*. Wiley Publishing Inc, 1st edition.

[Duggan, 1999] Duggan, D. (1999). Modular type-based reverse engineering of parameterized types in java code. *SIGPLAN Not.*, 34(10):97–113.

[ECMA, 2006] ECMA (2006). C# Language Specification. ECMA-334. Available at http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[ECMA, 2012] ECMA (2012). Common Language Infrastructure (CLI). Available at http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf.

[Ecma International, 2006] Ecma International (2006). Eiffel: Analysis, Design and Programming Language. Available at http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf.

[Eliens, 2000] Eliens, A. (2000). *Principles of Object-Oriented Software Development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[Elmasri and Navathe, 2010] Elmasri, R. and Navathe, S. (2010). *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition.

[Emerson Murphy-Hill, 2008] Emerson Murphy-Hill, A. P. B. (2008). Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38 – 44.

[Fayad et al., 1999] Fayad, M., Schmidt, G., and Johnson, R. (1999). *Building Applications Frameworks: Object-oriented Foundations of Framework Design*. Wiley Computer Publishing.

[Fields et al., 2009] Fields, J., Harvie, S., and Fowler, M. (2009). *Refactoring: Ruby Edition*. Addison-Wesley professional Ruby series. Addison-Wesley, 1st edition.

[Fowler, 1999] Fowler, M. (1999). *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1st edition.

[Fowler, 2013] Fowler, M. (2013). Refactoring Home. Available at http://www.refactoring.com/.

[Frenzel, 2006] Frenzel, L. (2006). The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. Eclipse Magazine. Available at http://www.eclipse.org/articles/Article-LTK/ltk.html.

[Fuente de la Fuente, 2009] Fuente de la Fuente, L. (2009). Plugin de Refactorización para Eclipse 2.0. Master's thesis, Univesidad de Burgos.

[Fuente de la Fuente and Herrero Paredes, 2008] Fuente de la Fuente, S. and Herrero Paredes, E. (2008). Dynamic Refactoring Plugin for Eclipse. Master's thesis, Universidad de Burgos.

[Fuhrer and Opdyke, 2009] Fuhrer, R. and Opdyke, W. F. (2009). WRT'09 3rd Worskhop on Refactoring Tools. In *WRT'09 3rd Worskhop on Refactoring Tools*. In OOPSLA'09, 24th International Conference on Object Oriented Programming, Systems, Languages and Applications, Orlando, Florida, USA, Fuhrer, Robert and Opdyke, William F.

[Fuhrer et al., 2005] Fuhrer, R., Tip, F., Kieżun, A., Dolby, J., and Keller, M. (2005). Efficiently refactoring Java applications to use generic libraries. In *ECOOP'05, 19th European Conference Object-Oriented Programming*, pages 71–96, Glasgow, Scotland.

[G. Sunyé et al., 2001] G. Sunyé, Pollet, D., LeTraon, Y., and J.-M. Jézéquel (2001). Refactoring UML models. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 134–138. Springer-Verlag.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition.

[Garcia et al., 2007] Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J., and Willcock, J. (2007). An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205.

[Garcia et al., 2003] Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J. G., and Willcock, J. (2003). A comparative study of language support for generic programming. *SIGPLAN Not.*, 38(11):115–134.

[Géraud and Duret-Lutz, 2000] Géraud, T. and Duret-Lutz, A. (2000). Generic programming redesign of patterns. In *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000)*, pages 283–294, Irsee, Germany.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Golding, 2005] Golding, T. (2005). *Professional .NET 2.0 Generics*. Wrox, 1st edition.

[Gómez San Martín and Mediavilla Saiz, 2011] Gómez San Martín, M. and Mediavilla Saiz, I. (2011). Clasificación de Refactorizaciones: Dynamic Refactoring Plugin 3.0. Master's thesis, Universidad de Burgos.

[Gorts, 2008] Gorts, S. (2008). Refactoring Thumbnails. Expressing the Evolution of Object Oriented Designs. Available at http://web.archive.org/web/20090101183426/http://www.refactoring.be/thumbnails.html.

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 3rd edition.

[Gregor et al., 2006] Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A. (2006). Concepts: Linguistic support for generic programming in C++. In *SIGPLAN Notices*, OOPSLA '06, pages 291–310, New York, NY, USA. ACM Press.

[Hutchison et al., 1981] Hutchison, D., Kobsa, A., Kanade, T., Kleinberg, J., and Kittler, J. (1981). *Programming Language Ada: Reference Manual. Proposed Standard Document United States Department of Defense*, volume 106 of *Lecture Notes in Computer Science*. Springer.

[ISE, 1992] ISE (1992). LDL User's Manual. Technical Report TR-AN-2/UM Version 1.5, Interactive Software Engineering Inc. (ISE).

[ISO, 1996] ISO (1996). Information Technology - Syntactic Metalanguage - Extended BNF. ISO/IEC 14977.

[Izquierdo and Molina, 2009] Izquierdo, J. L. C. and Molina, J. G. (2009). Gra2MoL: Una herramienta para la Extracción de Modelos en Modernización de Software. In Vallecillo, A. and Sagardui, G., editors, *JISBD*, pages 162–165.

[JetBrains, 2013a] JetBrains (2013a). IntelliJ IDEA :: The Most Intelligent Java IDE. Available at http://www.jetbrains.com/idea/. Java IDE.

[JetBrains, 2013b] JetBrains (2013b). ReSharper:: The Most Intelligent Add-In to Visual Studio. Available at http://www.jetbrains.com/resharper/.

[Jézéquel et al., 1999] Jézéquel, J.-M., Train, M., and Mingins, C. (1999). *Design Patterns and Contracts*. Addison-Wesley, 1st edition.

[Johnson and Foote, 1988] Johnson, R. E. and Foote, B. (1988). Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35.

[Johnson and Opdyke, 1993] Johnson, R. E. and Opdyke, W. F. (1993). Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742, pages 264–278. Springer-Verlag.

[Joyner, 1999] Joyner, I. (1999). *Objects Unencapsulated : Java, Eiffel, and C++??* Prentice Hall, Inc, Upper Saddle River, NJ, USA, 1st edition.

[Kennedy and Syme, 2001] Kennedy, A. and Syme, D. (2001). Design and implementation of generics for the .NET Common Language Runtime. *SIGPLAN Not.*, 36(5):1–12.

[Kerievsky, 2004] Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley Professional, 1st edition.

[Kiezun et al., 2007] Kiezun, A., Ernst, M. D., Tip, F., and Fuhrer, R. M. (2007). Refactoring for Parameterizing Java Classes. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 437–446, Minneapolis, MN, USA.

[Kniesel and Koch, 2004] Kniesel, G. and Koch, H. (2004). Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51.

[Kunert, 2008] Kunert, A. (2008). Semi-automatic generation of metamodels and models from grammars and programs. In *Electronic Notes in Theoretical Computer Science*, volume 211, pages 111–119, Amsterdam, The Netherlands, The Netherlands. Elsevier Science Publishers B. V.

[Lämmel, 2002] Lämmel, R. (2002). Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, RULE '02, pages 15–28, New York, NY, USA. ACM Press. 14 pages.

[Lehman et al., 2000] Lehman, M., Ramil, J. F., and Kahen, G. (2000). Evolution as a Noun and Evolution as a Verb. In *Proc. Workshop on Software and Organisation Co-evolution (SOCE)*, Durham, UK. University of Durham, K. Bennett.

[Lehman, 1980] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076.

[Lehman, 1996] Lehman, M. M. (1996). Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, pages 108–124. Springer-Verlag.

[Lehman and Belady, 1985] Lehman, M. M. and Belady, L. A., editors (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1st edition.

[Lieberherr et al., 1988] Lieberherr, K., Holland, I., and Riel, A. (1988). Object-oriented programming: an objective sense of style. *SIGPLAN Not.*, 23(11):323–334.

[Lieberherr and Holland, 1989] Lieberherr, K. J. and Holland, I. M. (1989). Assuring Good Style for Object-Oriented Programs. *IEEE Software*, 6(5):38–48.

[Liskov, 1976] Liskov, B. (1976). Introduction to CLU. In Schuman, S. A., editor, *New Directions in Algorithmic Languages 1975*. INRIA.

[Liskov et al., 1981] Liskov, B., Atkinson, R. R., Bloom, T., Moss, J. E. B., Schaffert, C., Scheifler, R., and Snyder, A. (1981). *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer.

[Liskov et al., 1995] Liskov, B., Curtis, D., Day, M., Ghemawat, S., Gruber, R., Johnson, P., and Myers, A. C. (1995). *Theta Reference Manual.* Programming Methodology Group Memo 88. MIT Laboratory for Computer Science, Cambridge, MA 02139. Available at http://www.pmg.lcs.mit.edu/papers/thetaref/index.html.

[Liskov et al., 1998] Liskov, B., Mathewson, N., and Myers, A. (1998). Overview of PolyJ. Available at http://www.pmg.csail.mit.edu/polyj/overview.ps.

[López and Crespo, 2003] López, C. and Crespo, Y. (2003). 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. Available at http://giro.infor.uva.es/Publications/2003/LC03.

[López et al., 2003] López, C., Marticorena, R., and Crespo, Y. (2003). Hacia una solución basada en frameworks para la definición de refactorizaciones con independencia del lenguaje. In Ernesto Pimentel, Nieves R. Brisaboa, J. G., editor, *VIII Jornadas Ingeniería del Software y Bases de Datos (JISBD 2003), Alicante, Spain. ISBN 84-688-3836-5*, pages 251–262. Available at http://giro.infor.uva.es/Publications/2003/LMC03.

[López et al., 2004] López, C., Marticorena, R., and Crespo, Y. (2004). 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. Available at http://giro.infor.uva.es/Publications/2004/LMC04.

[López et al., 2006a] López, C., Marticorena, R., and Crespo, Y. (2006a). 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. Available at http://giro.infor.uva.es/Publications/2006/LMC06.

[López et al., 2006b] López, C., Marticorena, R., Crespo, Y., and Pérez, J. (2006b). Towards a Language Independent Refactoring Framework. In *1st ICSOFT 06 International Conference on Software and Data Technologies. Setubal, Portugal. ISBN: 972-8865-69-4*, volume 1, pages 165–170. Available at http://giro.infor.uva.es/Publications/2006/LMCP06.

[Madsen et al., 1993] Madsen, O. L., Mø-Pedersen, B., and Nygaard, K. (1993). *Object-oriented programming in the BETA programming language.* ACM Press Series. ACM Press/Addison-Wesley Publishing Co., 2nd edition.

[Markiewicz and de Lucena, 2001] Markiewicz, M. E. and de Lucena, C. J. P. (2001). Object oriented framework development. *Crossroads*, 7(4):3–9.

[Marticorena, 2005] Marticorena, R. (2005). Analysis and Definition of a Language Independent Refactoring Catalog. In *17th Conference on Advanced Information Systems Engineering (CAiSE 05). Doctoral Consortium, Porto, Portugal.*, pages 70–77. Available at http://www.giro.infor.uva.es/Publications/2005/Mar05.

[Marticorena and Crespo, 2003] Marticorena, R. and Crespo, Y. (2003). Refactorizaciones de Especialización sobre el Lenguaje Modelo MOON. Technical Report DI-2003-02, Departamento de Informática. Universidad de Valladolid. Available at http://giro.infor.uva.es/Publications/2003/MC03.

[Marticorena and Crespo, 2008] Marticorena, R. and Crespo, Y. (2008). Dynamism in Refactoring Construction and Evolution. A Solution Based on XML and Reflection. In Cordeiro, J., Shishkov, B., Ranchordas, A., and Helfert, M., editors, *ICSOFT 2008, Third International Conference on Software and Data Technologies, Porto, Portugal*, pages 214 – 219. INSTICC - Institute for Systems and Technologies of Information, Control and Communication. Available at http://giro.infor.uva.es/Publications/2008/MC08.

[Marticorena and Crespo, 2012] Marticorena, R. and Crespo, Y. (2012). ALF como Lenguaje de Especificación de Refactorizaciones. In Ruíz, A. and Iribarne, L., editors, *Actas de la XVII Jornadas de Ingeniería del Software y Bases de Datos*, ISBN: 978-84-15487-28-9, pages 255 – 268, Almería. Sistedes. Available at http://www.giro.infor.uva.es/Publications/2012/MC12/.

[Marticorena et al., 2008] Marticorena, R., Crespo, Y., and López, C. (2008). Refactorizaciones en la Migración del Software. In *Actas JISBD'08, XIII Jornadas de Ingeniería del Software y Bases de Datos, Gijón, 2008. ISBN:978-84-612-5820-8*, pages 409–414, Gijón, Spain. Available at http://giro.infor.uva.es/Publications/2008/MCL08.

[Marticorena et al., 2011a] Marticorena, R., Gómez, M., Mediavilla, I., and Crespo, Y. (2011a). Construcción de Refactorizaciones para Eclipse: Dynamic Refactoring Plugin 3.0. In *Actas de la XVI Jornadas de Ingeniería del Software y Bases de Datos*, ISBN: 978-84-9749-486-1, pages 495 – 498, La Coruña. Sistedes. Available at http://www.giro.infor.uva.es/Publications/2011/MGMC11/.

[Marticorena et al., 2003] Marticorena, R., López, C., and Crespo, Y. (2003). 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. Available at http://giro.infor.uva.es/Publications/2003/MLC03.

[Marticorena et al., 2004] Marticorena, R., López, C., and Crespo, Y. (2004). Estudio de la Distribución Docente de Pruebas del Software y Refactoring para la Incorporación de Metodologías 'Agiles. In *JENUI'04, X Jornadas de Enseñanza Universitaria en Informática, Alicante, España*, pages 247–254. Available at http://giro.infor.uva.es/Publications/2004/MLC04.

[Marticorena et al., 2005] Marticorena, R., López, C., and Crespo, Y. (2005). Parallel Inheritance Hierarchy: Detection from a Static View of the System. In *6th International Workshop on Object Oriented Reenginering (WOOR), Glasgow, UK.*, page 6. http://smallwiki.unibe.ch/woor/workshopparticipants/.

[Marticorena et al., 2006] Marticorena, R., López, C., and Crespo, Y. (2006). Extending a Taxonomy of Bad Code Smells with Metrics. In *7th ECCOP International Workshop on Object-Oriented Reengineering (WOOR). Nantes, France. http://smallwiki.unibe.ch/woor/*, page 6. Available at http://giro.infor.uva.es/Publications/2006/MLC06.

[Marticorena et al., 2007a] Marticorena, R., López, C., and Crespo, Y. (2007a). Definición de un Proceso para la Contrucción de Refactorizaciones. In *JISBD'07, XII Jornadas Ingeniería del Software y Bases de Datos, Zaragoza, Spain*, pages 361–367. Available at http://giro.infor.uva.es/Publications/2007/MLC07.

[Marticorena et al., 2011b] Marticorena, R., López, C., and Crespo, Y. (2011b). Afrontando la Evolución de los Lenguajes de Programación a través de Refactorizaciones. In *Actas de la XVI Jornadas de Ingeniería del Software y Bases de Datos*, ISBN: 978-84-9749-486-1, pages 277 – 290, La Coruña. Sistedes. Available at http://giro.infor.uva.es/Publications/2011/MLC11.

[Marticorena et al., 2010a] Marticorena, R., López, C., Crespo, Y., and Pérez, F. J. (2010a). Refactoring Generics in JAVA: a case study on Extract Method. In Rafael Capilla, J. C. D. n. and Ferenc, R., editors, *14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 217 – 226. IEEE Computer Society. Available at http://giro.infor.uva.es/Publications/2010/MLCP10.

[Marticorena et al., 2007b] Marticorena, R., López, C., Crespo, Y., and Pérez, J. (2007b). Reuse Based Refactoring Tools. In *Proceedings 1st Workshop on Refactoring Tools (WRT 07).*, pages 21–23. ECOOP'07. Available at http://giro.infor.uva.es/Publications/2007/MLCP07.

[Marticorena et al., 2011c] Marticorena, R., López, C., Pérez, J., and Crespo, Y. (2011c). Assisting Refactoring Tool Development through Refactoring Characterization. In Maria Jose Escalona, B. S. and Cordeiro, J., editors, *6th International Conference on Software and Data Technologies (ICSOFT 2011) ISBN: 978-989-8425-77-5*, volume 2, pages 232 – 237. SciTePress - Science and Technology Publications. Available at http://giro.infor.uva.es/Publications/2011/MLPC11.

[Marticorena et al., 2010b] Marticorena, R., López, C., Crespo, Y., and Manso, E. (2010b). Umbrales relativos. caso de estudio para incorporar métricas en la detección de bad smells. *Revista de Procesos y Métricas*, 7(1):6–14. Available at http://giro.infor.uva.es/Publications/2010/MLCM10.

[McLaughlin and Flanagan, 2004] McLaughlin, B. D. and Flanagan, D. (2004). *Java 1.5 Tiger. A Developer's Notebook*. O'Reilly, 1st edition.

[Méndez et al., 2010] Méndez, M., Overbey, J., Garrido, A., Tinetti, F. G. T., and Johnson, R. (2010). A Catalog and Classification of Fortran Refactorings. In *ASSE 2010 - 11th Argentine Symposium on Software Engineering*, pages 501–505.

[Mendoça et al., 2004] Mendoça, N. C., Maia, P. H. M., Fonseca, L. A., and Andrade, R. M. C. (2004). RefaX: A Refactoring Framework Based on XML. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 147 – 156. IEEE Computer Society.

[Mens, 1999] Mens, T. (1999). *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Vrije Universiteit Brussel - Faculty of Sciences - Deparment of Computer Science - Programming Technology Lab.

[Mens et al., 2002] Mens, T., Buckley, J., Rashid, A., and Zenger, M. (2002). Towards a taxonomy of software evolution. Technical Report vub-prog-tr-02-05, Vrije Universiteit Brussel. Position paper at Workshop on Unanticipated Software Evolution, Warshau, Poland, 2003.

[Mens and Lanza, 2002] Mens, T. and Lanza, M. (2002). A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2):69–80.

[Mens and Tourwé, 2004] Mens, T. and Tourwé, T. (2004). A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139.

[Metsker, 2004] Metsker, S. J. (2004). *Design Patterns C#*. Addison-Wesley Professional, 1st edition.

[Meyer, 1988] Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall International, Upper Saddle River, NJ, USA, 1st edition.

[Meyer, 1990] Meyer, B. (1990). *Introduction to the Theory of Programming Languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition.

[Meyer, 1992] Meyer, B. (1992). *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, Upper Saddle River, NJ, USA.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition.

[Meyer, 2009] Meyer, B. (2009). *Touch of Class - Learning to Program Well with Objects and Contracts*. Springer-Verlag, 1st edition.

[Microsoft, 2010] Microsoft (2010). C# Language Specification Version 4.0. Available at http://www.microsoft.com/download/en/details.aspx?id=7029.

[Mills, 1971] Mills, H. (1971). *Chief programmer teams: principles and procedures*. Federal Systems Division, FSC 71-5108. IBM International Business Machines Corporation Report, Gaithersburg, Md.

[Monteiro and Fernandes, 2005] Monteiro, M. P. and Fernandes, J. a. M. (2005). Towards a Catalog of Aspect-Oriented Refactorings. In *Proceedings of the 4th international conference*

on Aspect-oriented software development, AOSD '05, pages 111–122, New York, NY, USA. ACM Press.

[Montmollin, 2013] Montmollin, G. d. (2013). The Transparent Language Popularity Index. Available at http://lang-index.sourceforge.net/.

[Musser and Stepanov, 1998a] Musser, D. and Stepanov, A. (1998a). Generic Programming. Dagstuhl Seminar on Generic Programming. Available at http://www.informatik.uni-trier.de/~ley/db/conf/dagstuhl/generic1998.html.

[Musser and Stepanov, 1998b] Musser, D. and Stepanov, A. (1998b). Generic Programming. Project and Open Problems. Available at http://www.cs.rpi.edu/~musser/gp/pop/.

[Musser and Stepanov, 1988] Musser, D. and Stepanov, A. A. (1988). Generic Programming. In *Symbolic and algebraic computation: ISSAC '88*, pages 13–25. Springer.

[Myers et al., 1997] Myers, A. C., Bank, J. A., and Liskov, B. (1997). Parameterized Types for Java. In *POPL'97, 24th Symposium on Principles of Programming Languages, Paris, France*, POPL '97, pages 132–145, New York, NY, USA. ACM.

[Naftalin and Wadler, 2007] Naftalin, M. and Wadler, P. (2007). *Java Generics and Collections*. O'Reilly Media, 1st edition.

[Neill and Laplante, 2006] Neill, C. J. and Laplante, P. A. (2006). Paying Down Design Debt with Strategic Refactoring. *Computer*, 39(12):131–134.

[Nierstrasz et al., 2005] Nierstrasz, O., Ducasse, S., and Gĭrba, T. (2005). The Story of Moose: An Agile Reengineering Environment. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Fo undations of software engineering*, ESEC/FSE-13, pages 1–10, New York, NY, USA. ACM.

[OMG, 2008] OMG (2008). Software & Systems Process Engineering Meta-Model (SPEM 2.0) Version 2.0. Technical Report formal/2008-04-01, Object Management Group. Available at http://www.omg.org/spec/SPEM/2.0/.

[OMG, 2010] OMG (2010). Action Language for Foundational UML (Alf). Concrete Syntax for a UML Action Language - Beta 1. Available at http://www.omg.org/spec/ALF/.

[OMG, 2011] OMG (2011). Semantics of A Foundational Subset for Executable UML Models (fUML) version 1.0. Available at http://www.omg.org/spec/FUML/.

[OMG, 2012] OMG (2012). Action Language for Foundational UML (Alf). Concrete Syntax for a UML Action Language - Beta 2. Available at http://www.omg.org/spec/ALF/.

[Omnicore, 2007a] Omnicore (2007a). CodeGuide.

[Omnicore, 2007b] Omnicore (2007b). X-Develop - Multi-language cross-platform IDE.

[Opdyke, 1992] Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA.

[Opdyke and Roberts, 1995] Opdyke, W. F. and Roberts, D. (1995). Refactoring. Tutorial presented at OOPSLA '95. In *10th International Conference on Object-Oriented Programming Systems, Languages and Applications*.

[Oracle, 2011] Oracle (2011). JDK 7 Features. http://openjdk.java.net/projects/jdk7/features/. Available at http://openjdk.java.net/projects/jdk7/features/.

[Overbey and Johnson, 2009] Overbey, J. L. and Johnson, R. E. (2009). Regrowing a language: refactoring tools allow programming languages to evolve. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 493–502, New York, NY, USA. ACM.

[Pirkelbauer et al., 2010] Pirkelbauer, P., Dechev, D., and Stroustrup, B. (2010). Source code rejuvenation is not refactoring. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '10, pages 639–650, Berlin, Heidelberg. Springer-Verlag.

[RefactorIt, 2007] RefactorIt (2007). RefactorIt - Aqris Software.

[Roberts, 1999a] Roberts, D. (1999a). Refactoring Browser. Available at http://st-www.cs.illinois.edu/users/brant/Refactory/.

[Roberts, 1999b] Roberts, D. B. (1999b). *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA.

[Shalloway and Trott, 2004] Shalloway, A. and Trott, J. (2004). *Design Patterns Explained: a new Perspective on Object-Oriented Design*. Addison-Wesley Professional, Boston, Mass., 2nd edition.

[Siff and Reps, 1996] Siff, M. and Reps, T. (1996). Program generalization for software reuse: from C to C++. *SIGSOFT Softw. Eng. Notes*, 21(6):135–146.

[Silberschatz et al., 2010] Silberschatz, A., Korth, H. F., and Sudarshan, S. (2010). *Database System Concepts*. McGraw-Hill Book Company, 6th edition.

[Sommerlad, 2012] Sommerlad, P. (2012). WRT'12 5th Worskhop on Refactoring Tools. In *WRT'12 5th Worskhop on Refactoring Tools*. In ICSE'12, 34th International Conference on Software Engineering, Zurich, Switzerland.

[Sommerville, 2010] Sommerville, I. (2010). *Software Engineering*. Addison-Wesley, 9th edition.

[Strein et al., 2006] Strein, D., Kratz, H., and Löwe, W. (2006). Cross-Language Program Analysis and Refactoring. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM'06, pages 207–216, Washington, DC, USA. IEEE Computer Society.

[Stroustrup, 1994] Stroustrup, B. (1994). *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1st edition.

[Stroustrup, 1997] Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley Professional, 3rd edition.

[Sweet, 1985] Sweet, R. E. (1985). The Mesa programming environment. *SIGPLAN Not.*, 20(7):216–229.

[Tichelaar, 1999] Tichelaar, S. (1999). FAMIX Java Language Plug-in 1.0. Available at http://scg.unibe.ch/archive/famoos/FAMIX/Plugins/JavaPlugin1.0.pdf.

[Tichelaar, 2001] Tichelaar, S. (2001). *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern.

[Tichelaar et al., 2000] Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O. (2000). A Meta-Model for Language-Independent Refactoring. In *Proc. International Workshop on Principles of Software Evolution (IWPSE)*, pages 157–169. IEEE Computer Society Press.

[TIOBE Software BV, 2013] TIOBE Software BV (2013). TIOBE Software: Tiobe Index. Available at http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[Tip et al., 2004] Tip, F., Fuhrer, R., Dolby, J., and Kiezun, A. (2004). Refactoring Techniques for Migrating Applications to Generic Java Container Classes. Technical Report Research Report RC 23238, IBM T.J. Watson Research Center.

[Tip et al., 2003] Tip, F., Kiezun, A., and Baumer, D. (2003). Refactoring for Generalization using Type Constraints. In *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 13–26, New York, NY, USA. ACM Press.

[Troelsen, 2001] Troelsen, A. (2001). *C# and the .NET Platform*. Apress.

[Van Gorp et al., 2003a] Van Gorp, P., Stenten, H., Mens, T., and Demeyer, S. (2003a). Towards Automating Source-Consistent UML refactorings. In Stevens, P., Whittle, J., and Booch, G., editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 144–158.

[Van Gorp et al., 2003b] Van Gorp, P., Van Eetvelde, N., and Janssens, D. (2003b). Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Meta model. In *Proceedings of 1st Fujaba Days*, University of Kassel (Germany).

[Visser, 2001] Visser, E. (2001). A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143.

[von Dincklage and Diwan, 2004] von Dincklage, D. and Diwan, A. (2004). Converting Java classes to use generics. In Vlissides, J. M. and Schmidt, D. C., editors, *OOPSLA'04, 19th International Conference on Object Oriented Programming, Systems, Languages and Applications, Vancouver, British Columbia, Canada*, pages 1–14. ACM.

[Wake, 2003] Wake, W. C. (2003). *Refactoring Workbook*. Addison-Wesley Professional, 1st edition.

[Whole Tomato Software, 2013] Whole Tomato Software, I. (2013). Visual Assist X for Visual Studio. Available at http://www.wholetomato.com/.

[Wimmer and Kramler, 2006] Wimmer, M. and Kramler, G. (2006). Bridging Grammarware and Modelware. In Bruel, J.-M., editor, *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, volume 3844 of *MoDELS'05*, pages 159–168, Berlin, Heidelberg. Springer-Verlag.

[Xref-Tech, 2007] Xref-Tech (2007). Xrefactory for C and C++ - A C/C++ Refactoring Browser for Emacs and XEmacs. Available at http://www.xref.sk/xrefactory/main.html.

[Zannier and Maurer, 2003] Zannier, C. and Maurer, F. (2003). Tool Support for Complex Refactoring to Design Patterns. In Marchesi, M. and Succi, G., editors, *Proceedings of the 4th international conference on Extreme Programming and agile processes in software engineering*, volume 2675 of *Lecture Notes in Computer Science*, pages 123–130, Berlin, Heidelberg. Springer-Verlag.