UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA
Escuela Técnica Superior de Ingeniería Informática
Departamento de Ingeniería de Software y Sistemas Informáticos

METODOLOGÍA DE DESARROLLO DE SOFTWARE BASADA EN EL PARADIGMA GENERATIVO. REALIZACIÓN MEDIANTE LA TRANSFORMACIÓN DE EJEMPLARES

TESIS DOCTORAL

Rubén Heradio Gil
Licenciado en Informática
2007

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA
Escuela Técnica Superior de Ingeniería Informática
Departamento de Ingeniería de Software y Sistemas Informáticos

METODOLOGÍA DE DESARROLLO DE SOFTWARE BASADA EN EL PARADIGMA GENERATIVO. REALIZACIÓN MEDIANTE LA TRANSFORMACIÓN DE EJEMPLARES

Memoria que presenta para obtener el grado de Doctor

Rubén Heradio Gil
Licenciado en Informática por la Universidad Politécnica de Madrid

Directores:

José Antonio Cerrada Somolinos
Catedrático de Universidad del Departamento de Ingeniería de Software y Sistemas Informáticos de la Universidad Nacional de Educación a Distancia

José Félix Estívariz López
Profesor Titular de Escuela de Universitaria del Departamento de Ingeniería de Software y Sistemas Informáticos de la Universidad Nacional de Educación a Distancia

Agradecimientos

Me siento en deuda con los autores que cito en la bibliografía, auténticos gigantes sobre cuyos hombros he desarrollado esta tesis. En especial, con Eugenio Arellano.

Agradezco a José Antonio Cerrada y a José Félix Estívariz, directores de esta tesis, sus comentarios y el entusiasmo que han puesto en este trabajo.

A mis compañeros, Juan José Escribano, S. Rubén Gómez, José Luis Gayo, Carlos Vicente Álvarez, Juan Antonio Mascarell, Ismael Abad y David Fernández, les agradezco su apoyo técnico y moral.

A Mari Carmen Cuéllar, María Gil, Juan Antonio Heradio, Israel Heradio, Jesús Muñoz y Gerardo Ramos, les agradezco su "soporte afectivo".

Muchas gracias.

Pasaba largas horas en su cuarto, haciendo cálculos sobre las posibilidades estratégicas de su arma novedosa, hasta que logró componer un manual de una asombrosa claridad didáctica y un poder de convicción irresistible.

Gabriel García Márquez, *Cien años de soledad.*

Resumen

Muchos autores consideran que el desarrollo de familias de productos, frente a la construcción individual de productos aislados, es un paso decisivo hacia la reutilización sistemática de software y la obtención de economía de alcance. Esta tesis se adscribe a esta corriente y propone un nuevo proceso de desarrollo de familias de productos, denominado EDD (*Exemplar Driven Development*), que aprovecha la similitud entre los productos de una familia para construirlos por analogía.

La primera actividad de EDD es la realización de un producto concreto de una familia. A continuación, se busca cómo flexibilizar este ejemplar para que satisfaga los requisitos del resto de los productos. Es decir, se trata de definir formalmente una relación de analogía que permita derivar del ejemplar los demás productos de forma automática. Por último, se obtienen los productos de la familia parametrizando la flexibilización del ejemplar.

Entre las aportaciones de EDD, cabe destacar:

- Abordar el desarrollo y el mantenimiento de una familia de productos mediante una estrategia sistemática e iterativa. Lo primero que se construye es un ejemplar que satisface los requisitos fijos de la familia. Después, se incorporan progresivamente capas de flexibilización que implementan los requisitos variables.

- Los requisitos fijos de una familia de productos suelen ser más estables que los requisitos variables. EDD separa la implementación de los requisitos fijos (el ejemplar) de la implementación de los requisitos variables (los módulos que flexibilizan el ejemplar).

- La decisión de elaborar una familia a menudo se toma al detectar trabajo repetitivo en el desarrollo aislado de varios productos de un dominio o al identificar oportunidades de negocio en la ampliación de las prestaciones de un producto de éxito. EDD reconoce esta situación y trata de aprovecharla mediante la reutilización íntegra de un ejemplar.

La tesis explora distintas maneras de flexibilizar un ejemplar aplicando las técnicas más comunes de generalización de código (herencia, genericidad, plantillas de código…).

Lamentablemente, se demuestra que estas técnicas padecen limitaciones que impiden flexibilizaciones satisfactorias (que sean modulares, no invasivas, aplicables a cualquier producto software…). Para superar estas limitaciones, la tesis propone el lenguaje original ETL (*Exemplar Transformation Language*), del que incluye una implementación en Ruby.

Con el fin de ilustrar la potencia y versatilidad de EDD y ETL, la tesis contiene ejemplos de desarrollo de programas escritos en Java y C++; de procedimientos almacenados escritos en TRANSACT SQL; de juegos de prueba escritos en Java y Modula-2; y de documentación escrita en HTML y Javadoc.

Abstract

Many authors consider that product family development, instead of one-off development, is a decisive step towards the systematic software reuse and economy of scope. This thesis is aligned with this movement and proposes a new process to build product families, named EDD (*Exemplar Driven Development*), which takes advantage of the similarity between family products to make them by analogy.

The first EDD activity is to build a specific product of a family. Next, this exemplar is flexibilized to satisfy the remaining products requirements. That is, an analogy relation is defined in a formal way to derive products automatically from the exemplar. Finally, the family products are obtained parametrizing the exemplar flexibilization.

Among EDD contributions should be mentioned the following:

- Facing the development and maintenance of a product family with an iterative strategy. An exemplar which satisfies the family fixed requirements is built as soon as possible. Flexibilization layers, which implement family variable requirements, are added to the exemplar in successive development cycles.

- Family fixed requirements are usually more stable than variable requirements. EDD separates the fixed requirements implementation (located in the exemplar) from the variable requirements implementation (located in the modules which make the exemplar flexible).

- Building a product family is usually decided when repetitive work is detected in the development of specific products in the same domain or when business opportunities are expected if a successful product is extended. EDD recognizes this situation and tries to take advantage of it by reusing the exemplar completely.

This thesis explores different ways to flexibilize an exemplar by applying the most common techniques of code generalization (inheritance, genericity, code templates…). Unfortunately, the thesis demonstrates that these techniques suffer limitations which hinder flexibilizations with important qualities like modularity, non-invasiveness, applicability to any software product… In order to avoid these limitations, the thesis

proposes a new language called ETL (*Exemplar Transformation Language*) which is implemented in Ruby.

With the purpose of illustrating the power and versatility of EDD and ETL, the thesis includes examples of the development of programs written in Java and C++; stored procedures written in TRANSACT SQL; tests suites written in Java and Modula-2; and documentation written in HTML and Javadoc.

Índice general

1. Introducción

1.1. Justificación del desarrollo de familias de productos 25
1.2. Procesos de desarrollo de familias de productos 27
1.3. Proceso EDD de desarrollo de familias de productos 29
1.4. Lenguaje de transformaciones ETL 30
1.5. Organización de la tesis 33

2. Estado del arte

2.1. La reutilización de software 36
2.1.1. La reutilización en el análisis 36
2.1.2. La reutilización en el diseño 39
2.1.2.1. Diseño arquitectónico 39
2.1.2.2. Patrones de diseño 41
2.1.2.3. Marcos de trabajo 46
2.1.3. La reutilización en la codificación 48
2.1.3.1. Programación orientada a aspectos 49
2.1.4. La reutilización en las pruebas 53
2.2. Desarrollo de familias de productos 53
2.2.1. La programación generativa 53
2.2.2. Las factorías de software 57
2.2.3. El desarrollo dirigido por modelos 61
2.2.4. Resumen comparativo 66
2.3. Compiladores de DSLs 67
2.3.1. Lenguajes de transformación 72

3. Proceso EDD para el desarrollo de familias de productos

3.1. Construcción de un ejemplar 76
3.2. Flexibilización del ejemplar 77
3.2.1. Análisis de la familia de productos 77

3.2.1.1. Definición del dominio 77
3.2.1.2. Identificación y clasificación de los requisitos de la familia de productos 78
3.2.1.3. Modelado del dominio 79
3.2.1.4. Análisis de la rentabilidad de la flexibilización del ejemplar 79
3.2.1.5. Ajuste del modelo del domino 79
3.2.1.6. Selección de los requisitos de un ejemplar 79
3.2.2. Definición de una interfaz para la especificación abstracta de los productos 80
3.2.2.1. Especificación de un DSL 80
3.2.2.1.1. Sintaxis 80
3.2.2.1.2. Semántica 83
3.2.2.2. Cualidades deseables de un DSL 83
3.2.3. Implementación de la flexibilización del ejemplar 84
3.2.4. Flexibilización iterativa 87
3.3. Obtención de productos a partir de la flexibilización del ejemplar 88

4. Lenguaje de transformaciones ETL

4.1. Especificación de ETL 90
4.1.1. Primitivas 90
4.1.2. Operadores para la combinación de generadores 91
4.1.3. Metamodelo simplificado 91
4.1.4. Ejemplo 92
4.1.4.1. Definición de la interfaz para parametrizar la flexibilización 96
4.1.4.2. Implementación de la flexibilización 96
4.2. Implementación de ETL 98
4.2.1. Justificación de la implementación de ETL en Ruby 98
4.2.2. Implementación de las primitivas 100
4.2.3. Implementación de los operadores para la combinación de generadores 101
4.2.3. Ejemplo 103
4.2.3.1. Definición de la interfaz para parametrizar la flexibilización 103
4.2.3.2. Implementación de la flexibilización 104

5. Estudio comparativo de metodologías y técnicas de desarrollo de familias de productos

5.1. Ejemplo 1: "Desarrollo de diccionarios en Java" 110
5.1.1. Desarrollo de una infraestructura sin ocultación 111
5.1.2. Desarrollo de una infraestructura con ocultación 113

5.1.2.1. Infraestructura implementada como un intérprete 113
5.1.2.2. Generación de productos desde cero, escribiendo sobre un flujo de texto 115
5.1.2.3. Flexibilización de un ejemplar con ETL.....116
5.1.2.4. Comparación entre la generación de productos con ETL y la generación desde cero, escribiendo sobre un flujo de texto.....118
5.2. Ejemplo 2: "Desarrollo de una familia de listas para C++".....119
5.2.1. Resolución mediante GP.....120
I. Análisis del dominio 120
II. Diseño arquitectónico del dominio 121
III. Diseño detallado del dominio.....124
IV. Codificación de los componentes.....125
V. Ingeniería de aplicación 128
5.2.2. Resolución mediante EDD y ETL.....128
I. Análisis de la familia de productos 128
II. Construcción de un ejemplar de la familia de productos 128
III. Definición de una interfaz para parametrizar la flexibilización del ejemplar.....130
IV. Implementación de la flexibilización del ejemplar.....131
5.2.3. Aplicabilidad de las técnicas comunes de generalización de código a la flexibilización de un ejemplar.....134
5.2.3.1. Sentencia de selección 135
5.2.3.2. Subprogramas 137
5.2.3.3. Composición.....139
5.2.3.4. Composición y genericidad.....142
5.2.3.5. Herencia múltiple 145
5.2.3.6. Herencia simple.....147
5.2.3.7. Herencia parametrizada.....150
5.2.3.8. Orientación a aspectos 153
5.2.3.9. Plantillas de código 155
5.3. Conclusiones.....157

6. Ejemplos de aplicación

6.1. Documentación de código.....162
6.1.1. Análisis de la familia de productos.....163
6.1.2. Construcción de un ejemplar.....163
6.1.3. Definición de una interfaz para la especificación abstracta de los productos 165

6.1.4. Implementación de la flexibilización del ejemplar.....168
6.2. Lectura de datos.....171
6.2.1. Análisis de la familia de productos.....173
6.2.2. Definición de una interfaz para la especificación abstracta de los productos 175
6.2.3. Implementación de la flexibilización del ejemplar.....176
6.3. Copia de seguridad de una base de datos.....181
6.3.1. Análisis de la familia de productos.....192
6.3.2. Definición de una interfaz para la especificación abstracta de los productos 192
6.3.3. Implementación de la flexibilización del ejemplar.....194
6.4. Prueba de unidades.....197
6.4.1. Manual de usuario de m2unit.....198
6.4.2. Implementación de m2unit.....200
6.5. Aumento de la expresividad y la concisión de un lenguaje.....207
6.5.1. Interfaz de la extensión.....208
6.5.2. Preprocesador ETL.....208

7. Conclusiones y trabajo futuro

7.1. Conclusiones.....211
7.2. Trabajo futuro.....213

Bibliografía……..…….215

Lista de símbolos

ASP	*Active Server Pages*
ATL	*Atlas Transformation Language*
BPSL	*Balanced Pattern Specification Language*
CIM	*Computation Independent Model*
CO_2P_2S	*Correct Object-Oriented Pattern-based Programming System*
CSV	*Comma Separated Value*
DOM	*Document Object Model*
DSL	*Domain Specific Language*
EBNF	*Extended Backus-Naur Form*
EDD	*Exemplar Driven Development*
ETL	*Exemplar Transformation Language*
FAST	*Family-Oriented Abstraction, Specification and Translation*
FODA	*Feature-Oriented Domain Analysis*
FOL	*First Order Logic*
GP	*Generative Programming*
GPL	*General Purpose Language*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
JSP	*Java Server Pages*
MDA	*Model Driven Architecture*
MDD	*Model Driven Development*
MOF	*Meta-Object Facility*
MRAM	*Method for Requirements Authoring and Management*
OCL	*Object Constraint Language*
ODM	*Organization Domain Modeling*
OMG	*Object Management Group*
PIM	*Platform Independent Model*
POAD	*Pattern-Oriented Analysis and Design*
PSM	*Platform Specific Model*
QVT	*Query/View/Transformation*
SAX	*Simple API for XML*
TLA	*Temporal Logic Actions*

TUG	*Tree Unified with Grammar*
UML	*Unified Modeling Language*
W3C	*World Wide Web Consortium*
XMI	*XML Model Interchange*
XML	*eXtensible Markup Language*
XSLT	*Extensible Stylesheet Language Transformations*

Índice de figuras

Figura 1.1. Proceso típico de desarrollo de familias de productos. 27
Figura 1.2. Proceso EDD de desarrollo de familias de productos. 30
Figura 1.3. Flexibilizaciones invasivas y no invasivas de un ejemplar. 32
Figura 2.1. Pila genérica y dos contenedores alternativos para almacenar sus elementos. 37
Figura 2.2, 2.3 y 2.4. Diagramas MRAM para la representación de dependencias entre requisitos 39
Figura 2.5. Catálogo de patrones arquitectónicos incluido en POSA1 40
Figura 2.6. Catálogo de patrones de diseño incluido en [GHJV94]. 41
Figura 2.7. Modelo generativo ofrecido por CO_2P_2S y Meta-CO_2P_2S para la encapsulación de patrones de diseño. 43
Figura 2.8. Ejemplo de diseño con solapamiento de patrones (tomado de [MSU98]) 44
Figura 2.9. Inversión del flujo de control y colisiones en los marcos de trabajo. 48
Figura 2.10. Código enredado 50
Figura 2.11. Código disperso 50
Figura 2.12. Resumen del proceso de desarrollo orientado a aspectos. 50
Figura 2.13. Arquitectura propuesta en [CE00] para modelos generativos. 54
Figura 2.14. Notaciones para diagramas de características. 56
Figura 2.15. Diagrama de características "roles en una organización". 56
Figura 2.16. Diagrama de clases UML "roles en una organización. 57
Figura 2.17. Representación mediante un grafo del esquema de una factoría de software propuesto por J. Greenfield [GS04] 59
Figura 2.18. Representación tabular del esquema de una factoría de software propuesto por J. Zachman [Zac04] 60
Figura 2.19. Principales procesos y productos de una factoría de software 61
Figura 2.20. Equivalencia terminológica entre la GP, las factorías de software y MDA. 67
Figura 2.21. Arquitectura de un compilador según [ASU90] 70
Figura 2.22. Compilador que utiliza código GPL como paso intermedio. 70
Figura 2.23. Implementación en Ruby de un compilador con DSL "interno" 71
Figura 2.24. Implementación en C++ de un compilador con DSL y GPL "internos" 71
Figura 2.25. Arquitectura de QVT. 73
Figura 3.1. Visión panorámica de EDD (notación: diagrama de actividades UML). 76
Figura 3.2. Flexibilización de un ejemplar (notación: diagrama de actividades UML). 77
Figura 3.3. Ejemplo de modelo de un dominio. 81

Figura 3.4. Modelo podado de un dominio. 81
Figura 3.5. Tabla de conversión FODA → EBNF. 82
Figura 3.6. Gramática independiente del contexto equivalente al modelo de la figura 3.4. 82
Figura 3.7. Ejemplo de especificación. 82
Figura 3.8. Árbol sintáctico correspondiente a la especificación de la figura 3.7. 82
Figura 3.9. Técnicas de flexibilización que se evaluarán en el capítulo 5. 85
Figura 3.10. Integración de EDD y ETL en el ciclo de vida en espiral. 87
Figura 4.1. Ejemplo de compilador resultado de flexibilizar un ejemplar con ETL. 90
Figura 4.2. Metamodelo simplificado de ETL. 92
Figura 4.3. Diseño del programa de reciclaje. 92
Figura 4.4. Trash.java. 93
Figura 4.5. Paper.java. 93
Figura 4.6. Aluminum.java. 93
Figura 4.7. Recycle.java. 94
Figura 4.8. RecycleTest.java. 94
Figura 4.9. Resultado de la ejecución de RecycleTest.java. 95
Figura 4.10. Gramática abstracta de un DSL para especificar programas de reciclaje de basura. 96
Figura 4.11. Ejemplo de especificación XML de un programa de reciclaje de basura. 96
Figura 4.12. Ejemplo de generación del programa objeto especificado en la figura 4.11. 98
Figura 4.13. Diseño detallado de ETL en Ruby. 103
Figura 4.14. Ejemplo de análisis. 103
Figura 4.15. Analizador. 104
Figura 4.16. Generador RecycleGen. 105
Figura 4.17. Generador TestGen. 106
Figura 4.18. Configuración y ejecución del compilador. 107
Figura 5.1. Versión reducida de un diccionario para la revisión ortográfica de documentos en castellano. 110
Figura 5.2. Flexibilización de la figura 5.1 sin ocultación. 111
Figura 5.3. Interfaz `Language`. 112
Figura 5.4. La especificación de cada diccionario implementará `Language`. 112
Figura 5.5. Especificación de un diccionario de inglés para la infraestructura de la figura 5.2. 112
Figura 5.6. Especificación de un diccionario de 112
Modula-2 para la infraestructura de la figura 5.2. 112
Figura 5.7. Gramática de un DSL para especificar diccionarios. 113
Figura 5.8. Especificación, según la gramática de la figura 5.7, de un diccionario de inglés. 113
Figura 5.9. Especificación, según la gramática de la figura 5.7, de un diccionario de Modula-2. 113

Figura 5.10. Arquitectura de un intérprete de diccionarios. 114
Figura 5.11. Codificación de un intérprete de diccionarios. 114
Figura 5.12. Arquitectura de un compilador de diccionarios. Generación desde cero, escribiendo sobre un flujo de texto. 115
Figura 5.13. Codificación de un compilador de diccionarios. Generación desde cero, escribiendo sobre un flujo de texto. 116
Figura 5.14. Diccionario de inglés generado por la infraestructura de la figura 5.13 a partir de la especificación de la figura 5.8. 116
Figura 5.15. Diccionario deModula-2 generado por la infraestructura de la figura 5.13 a partir de la especificación de la figura 5.9. 116
Figura 5.16. Arquitectura de un compilador de diccionarios. Generación con ETL por analogía. 117
Figura 5.17. Diseño detallado de un compilador de diccionarios. Generación con ETL por analogía. 117
Figura 5.18. Codificación de un compilador de diccionarios. Generación con ETL por analogía. 118
Figura 5.19. Gramática, equivalente a la de la figura 5.7, de un DSL interno a Ruby 118
Figura 5.20. Especificación, según la gramática de la figura 5.18, de un diccionario de inglés. 118
Figura 5.21. Ejecución, paso a paso, del análisis interno realizado por el intérprete de Ruby de la especificación de la figura 5.20. 119
Figura 5.22. Modelo FODA del dominio "listas para C++". 122
Figura 5.23. Correspondencias entre los nodos del diagrama de la figura 5.22, las responsabilidades que debe asumir la línea de productos y las categorías de componentes que deben desarrollarse. 122
Figura 5.24. Dependencias de uso entre las categorías de componentes. 123
Figura 5.25. Organización de las categorías de componentes en una arquitectura por capas. 123
Figura 5.26. Gramática GenVoca que especifica las posibles combinaciones entre componentes. 124
Figura 5.27. Diseño detallado, obtenido con la metodología propuesta en [CE00], de la línea de productos "listas para C++". 125
Figura 5.28. Codificación de la línea de productos "listas para C++" obtenida con la metodología propuesta en [CE00]. 127
Figura 5.29. Ejemplo de almacén de configuración. 128
Figura 5.30. Combinación de las clases de la figura 5.28, que satisface la especificación de la figura 5.29. 128
Figura 5.31. Equivalencia entre notaciones FODA. 129
Figura 5.32. Requisitos de la familia "listas para C++" seleccionados para el desarrollo de un ejemplar. 129

Figura 5.33. Ejemplar de la familia “listas para C++”. 130
Figura 5.34. Gramática de un DSL para especificar listas. 131
Figura 5.35. Especificación de una lista según la gramática de la figura 5.34. 131
Figura 5.36. Gramática, equivalente a la de la figura 5.34, de un DSL interno a Ruby. 131
Figura 5.37. Especificación de una lista según la gramática de la figura 5.36. 131
Figura 5.38. Diseño de la flexibilización ETL del ejemplar. 132
Figura 5.39. Codificación de la flexibilización ETL del ejemplar. 133
Figura 5.40. Análisis de las especificaciones DSL y ejecución coordinada de los generadores ETL. 134
Figura 5.41. Orden de aplicación de los generadores ETL. 134
Figura 5.42. Flexibilización del ejemplar con sentencias de selección. 136
Figura 5.43. Registro para especificar listas según la flexibilización de la figura 5.42. 136
Figura 5.44. Especificación de una lista, lograda parametrizando el registro de la figura 5.43. 136
Figura 5.45 Generalización del ejemplar con subprogramas. 138
Figura 5.46. Diseño de una flexibilización del ejemplar basada en la composición y la herencia de clases. 139
Figura 5.47. Codificación de una flexibilización del ejemplar basada en la composición y la herencia de clases. 142
Figura 5.48. Ejemplo de especificación de una lista para la flexibilización de la figura 5.42.142
Figura 5.49. Flexibilización del ejemplar basada en la composición y la herencia de clases.144
Figura 5.50. Ejemplo de especificación de una lista para la flexibilización de la figura 5.49.145
Figura 5.51. Diseño de una flexibilización del ejemplar basada en herencia múltiple. 145
Figura 5.52. Modificaciones necesarias para la flexibilización del ejemplar. 147
Figura 5.53. Diseño de una flexibilización del ejemplar basada en herencia simple. 148
Figura 5.54. Extracto de la codificación de una generalización del ejemplar basada en herencia simple. 150
Figura 5.55. Diseño de una flexibilización del ejemplar basada en herencia parametrizada.150
Figura 5.56. Codificación de una generalización del ejemplar basada en herencia parametrizada. 153
Figura 5.57. Diseño, según la notación propuesta en [Zha05], de una flexibilización del ejemplar basada en orientación a aspectos. 155
Figura 5.58. Flexibilización del ejemplar con plantillas ERB. 157
Figura 5.59. Análisis de especificaciones DSL y generación de listas con plantillas ERB. 157
Figura 6.1. Model del dominio “documentación de la creación de tablas en SQL”. 163
Figura 6.2. Representación en un navegador del ejemplar de SQLDoc. 164
Figura 6.3. Ejemplar de SQLDoc (index.html). 164
Figura 6.4.Ejemplar de SQLDoc (Tables.html). 164
Figura 6.5. Ejemplar de SQLDoc (Book.html). 165

Figura 6.6. Sintaxis abstracta de un DSL para documentar la creación de tablas en SQL. 165
Figura 6.7. Sintaxis concreta de un DSL para documentar la creación de tablas en SQL. 166
Figura 6.8. Ejemplo de código SQL documentado (banco.sql) 167
Figura 6.9. Generación obtenida por SQLDoc a partir de la figura 6.8 (Tables.html) 167
Figura 6.10. Generación obtenida por SQLDoc a partir de la figura 6.8 (Cuenta.html). .. 167
Figura 6.11. Arquitectura de SQLDoc 168
Figura 6.12. Diseño del generador SQLDoc. 169
Figura 6.13. Código de SQLDoc (SQLDoc.rb). 170
Figura 6.14. Ejemplo de fichero de datos de personas ("data.csv"). 171
Figura 6.15. Lector de datos de personas ("PeopleReader.java") 173
Figura 6.16. Modelo del dominio "lectores de ficheros CSV" 174
Figura 6.17. Documentación, generada con la herramienta javadoc, del lector de personas. 174
Figura 6.18. Juego de pruebas para el lector de personas ("PeopleTest.java") 175
Figura 6.19. Gramática EBNF del DSL con el que se especificarán los lectores de ficheros CSV. 176
Figura 6.20. Especificación XML de un lector de libros. 176
Figura 6.21. Ejemplo de fichero de datos para el lector especificado en la figura 6.20 176
Figura 6.22. Diseño del compilador de "lectores de ficheros CSV". 177
Figura 6.23. Analizador de las especificaciones de programas lectores de datos. 177
Figura 6.24. Ejemplo de generación del programa objeto especificado en la figura 6.20. . 178
Figura 6.25. Generador ReaderGen. 179
Figura 6.26. Generador TestGen. 180
Figura 6.27. Módulo auxiliar CSV_Utilities. 180
Figura 6.28. Suma de los generadores ReaderGen y TestGen. 181
Figura 6.29. Copia de datos entre tablas originales y temporales. 181
Figura 6.30. Tabla de seguimiento de la copia de datos. 182
Figura 6.31. Procedimientos almacenados para la copia de los datos contenidos en la tabla TbCompProp. 182
Figura 6.32. Procedimiento almacenado sp_AprovTbCompProp. 187
Figura 6.33. Procedimiento almacenado sp_ActuTbDiaCtlPrc. 188
Figura 6.34. Procedimiento almacenado sp_ActuTbCompProp. 190
Figura 6.35. Procedimiento almacenado sp_ExisTbCompProp. 191
Figura 6.36. Modelo del dominio "copiadores de datos entre tablas". 192
Figura 6.37. Diseño del compilador de "copiadores de datos entre tablas". 193
Figura 6.38. "Análisis" de la especificación de los procedimientos almacenados para las tablas TbAtrTb y TbTb. Ejecución del generador AprovTbGen. 194
Figura 6.39. Ejemplo de generación de los procedimientos almacenados que copian los datos de la tabla TbAtrTb. 194
Figura 6.40. Generador AprovTbGen. 197

Figura 6.41.Módulo Util. 198
Figura 6.42. Módulo Util enriquecido con las pruebas Test1 y Test2. 199
Figura 6.43. Resultado de ejecutar con m2unit la figura 6.42. 200
Figura 6.44. Ejemplo de actuación de los generadores de m2unit. 200
Figura 6.45. Módulo principal Test1 generado por m2unit. 202
Figura 6.46. Módulo principal Test2 generado por m2unit. 203
Figura 6.47. Código de m2unit. 207
Figura 6.48. Programa Java enriquecido con la notación aritmética para la clase BigDecimal. 207
Figura 6.49. Programa Java convencional equivalente a la figura 6.48. 208
Figura 6.50. Preprocesador BigDecEqGen escrito en ETL. 209
Figura 6.51. Compilador BigDecEqParser escrito en Racc. 210

1

Introducción

Nothing is really unprecedented. Faced with a new situation, people liken it to familiar ones and shape their response on the basis of the perceived similarities.

M.S. Mahoney, *The Roots of Software Engineering.*

1.1. Justificación del desarrollo de familias de productos

A finales de los 60, M. D. McIlroy [McI68] señaló la relevancia de la **reutilización** como factor decisivo para mejorar la calidad del software y reducir los costes de desarrollo y mantenimiento. Desde entonces, se han sucedido numerosos intentos para posibilitar y mejorar la reutilización de todo tipo de productos software [BG93, Mil95, LL97, KSBM99, Sel05, Tra94].

I. Sommerville [Som05] resume los beneficios de la reutilización de software en los siguientes puntos:

- Reducción de costes.
- Desarrollo acelerado.
- Aumento de la confiabilidad. El software reutilizado, frente al de nueva construcción, tiene la ventaja de haber probado su validez en otros desarrollos.

- Reducción de riesgos en el proceso de desarrollo. La incertidumbre sobre el coste de desarrollo se limita a los nuevos artefactos, lo que reduce el margen de error en la estimación de los costes.

- Uso efectivo de los especialistas. La reutilización libera a los desarrolladores del trabajo repetitivo y permite que dediquen su tiempo a tareas más novedosas y estimulantes.

Aunque algunas estimaciones realizadas en los años 80 [LG84] pronosticaban que el 60% de una aplicación informática se desarrollaría ensamblando componentes reutilizables, el nivel de reutilización alcanzado hoy día es bastante inferior. Por ejemplo, un estudio publicado en 2005 [Sel05] revelaba que el porcentaje de reutilización logrado en 25 proyectos de la NASA con 3.000-112.000 líneas de código era del 32%.

Muchos autores [CE00, Nei80, Par76] consideran que este fracaso se debe a que la mayoría de los procesos de desarrollo de software, ya sean formales o ágiles, persiguen la construcción de productos aislados (*one-off development).* Al no disponerse de contextos suficientemente amplios como para detectar con precisión qué elementos son reutilizables y cuáles son las situaciones donde puede sacarse más partido a la reutilización, se desemboca en una reutilización oportunista del software. Para que la reutilización del software fuera sistemática, los procesos de desarrollo deberían abordar la construcción colectiva de **familias de productos**[1] relacionados por un dominio.

Otros autores [GS04] han llegado a conclusiones similares al tratar de aplicar en la fabricación de software los principios de economía de escala y de alcance (*economies of scale and scope*), comúnmente utilizados en la industria para reducir los costes y tiempos de fabricación y mejorar la calidad de los productos.

La **economía de escala** se refiere a la fabricación de múltiples unidades de un mismo producto. Cuanto más se produce, menores son los costes. Se logra por diversas causas: reparto de los costes fijos entre más unidades producidas (disminución del coste medio), *rappel* sobre compras, mejora tecnológica, incremento de la racionalidad en el trabajo (especialización y división del trabajo)…

---

[1] D. Parnas [Par76] fue quien acuñó el término "familia de programas" para designar a un conjunto de programas lo suficientemente similares como para que sea más rentable resolverlos colectivamente que por separado: "*We consider a set of programs to constitute a family, whenever it is worth-while to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members.*"

La **economía de alcance** se da en la fabricación colectiva de productos similares. Se consigue principalmente porque los problemas comunes en la fabricación de los diversos productos se resuelven una sola vez.

La fabricación industrial de un producto consta fundamentalmente de dos etapas [GS04]:

- La fase de desarrollo, donde se crean el diseño del producto y unos pocos prototipos para la validación del diseño.

- La fase de producción, donde se crean de forma masiva instancias del producto.

La economía de escala ocurre sobre todo durante la fase de producción. Como señala P. Weger [Weg78], la naturaleza esencialmente lógica del software hace que los costes se concentren en la etapa de desarrollo (el coste de producir las copias de un sistema informático es despreciable comparado con el coste de desarrollo del sistema) y, por tanto, sea la economía de alcance el principio más aplicable en la fabricación de software.

En resumen, el desarrollo de familias de productos, frente a la construcción individual de productos aislados, es un paso decisivo hacia la reutilización sistemática de software y la obtención de economía de alcance.

1.2. Procesos de desarrollo de familias de productos

En los últimos tiempos, la tendencia a construir familias de productos ha ido en aumento y ha cristalizado en diversos procesos de desarrollo [CE00, GS04, MSUW04], cuyo esquema general se resume en la figura 1.1. Como se verá en la sección 2.2, estos procesos suelen descomponerse en dos grandes actividades: la realización de una infraestructura que implemente de manera global los requisitos de todos los productos de la familia y la obtención posterior de cada producto, parametrizando dicha infraestructura.

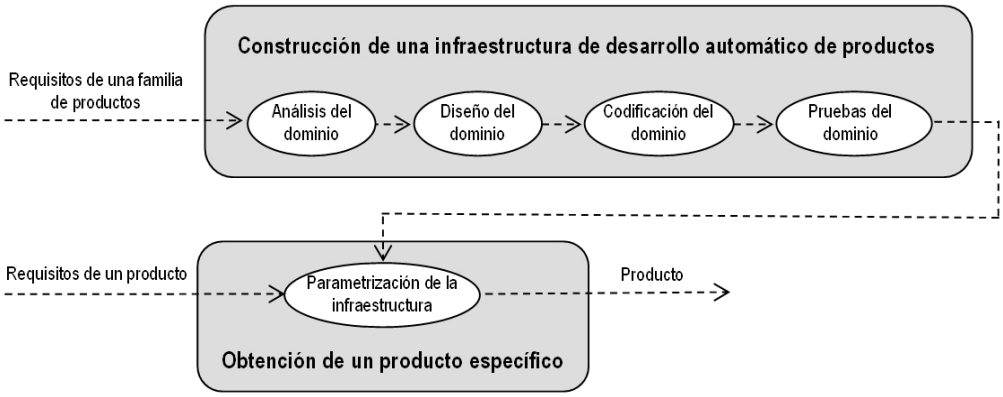


Figura 1.1. Proceso típico de desarrollo de familias de productos.

El desarrollo de la infraestructura suele descomponerse en una fase de "análisis de dominio", donde se especifican de forma colectiva los requisitos de los productos de una familia, y una fase de "implementación del dominio" (diseño, codificación y pruebas), donde se construye la infraestructura que satisface los requisitos obtenidos en la fase anterior.

Para el **análisis de dominio** se han creado nuevas metodologías, como FAST (*Family-Oriented Abstraction, Specification and Translation*) [WL99], ODM (*Organization Domain Modeling*) [ODM06], FODA (*Feature-Oriented Domain Analysis*) [KCHN+90]…

Respecto a la **implementación del dominio**, el planteamiento inicial consistía en construir un conjunto de componentes capaces de satisfacer los requisitos comunes de todos los productos de una familia. La obtención posterior de cada producto se conseguía configurando y ensamblando manualmente dichos componentes.

De las librerías de componentes (*libraries, toolkits*) se evolucionó hacia los marcos de trabajo (*frameworks*). Según E. Gamma et al [GHJV94], "un marco de trabajo es un conjunto de clases cooperantes que constituyen un diseño reutilizable para un tipo específico de software". Un marco de trabajo, además de componentes reutilizables, proporciona un esqueleto común para los productos de una familia, lo que reduce considerablemente el esfuerzo de ensamblado de los componentes.

Actualmente, se considera que los marcos de trabajo son abstracciones de caja gris (*gray box abstractions*) [GS04] y como consecuencia,

- su reutilización exige un esfuerzo de aprendizaje considerable.
- se hace depender la evolución de los productos de la implementación del marco. Por ejemplo, si se han desarrollado interfaces gráficas de usuario con el marco de trabajo AWT de Java, la actualización de las interfaces al marco más moderno SWING implicará su reescritura.

Por ello, se prefiere la creación de infraestructuras que oculten los detalles de implementación mediante un **lenguaje específico de dominio** (**DSL**, *Domain Specific Language*). Estas infraestructuras actúan como compiladores que traducen automáticamente las abstractas especificaciones DSL en productos finales[2].

---

[2] Por abstracción suele entenderse el acto de quedarse con lo esencial prescindiendo de los detalles que, desde algún punto de vista, son irrelevantes. La abstracción:

1.3. Proceso EDD de desarrollo de familias de productos

Esta tesis propone un nuevo proceso de desarrollo, denominado **EDD** (*Exemplar Driven Development*), que explota la similitud entre los productos de una familia para obtenerlos por **analogía**[3] (si no existiera tal similitud, no tendría sentido abordar el desarrollo de los productos de forma colectiva).

La figura 1.2 resume el proceso EDD, que se descompone en tres actividades básicas:

1. La construcción de un producto concreto de la familia, al que nos referiremos como **ejemplar**.

2. La realización de una infraestructura con ocultación de caja negra, que facilite la obtención posterior de los productos. La infraestructura se desarrollará **flexibilizando** el ejemplar, es decir, definiendo la relación de analogía que permitirá derivar automáticamente del ejemplar todos los productos de la familia.

3. La obtención de cada producto parametrizando la infraestructura.

Frente a los procesos típicos de desarrollo de familias de productos, EDD ofrece interesantes ventajas. Por ejemplo:

- Aborda el desarrollo y el mantenimiento de una familia de productos mediante una estrategia sistemática e iterativa. Lo primero que se construye es un ejemplar

---

- Libera a los desarrolladores de preocupaciones superfluas y permite que centren su atención en las cuestiones fundamentales.
- Propicia la concisión en la especificación del software, mejorando la legibilidad de las especificaciones y, por tanto, su mantenimiento.

Fuera del contexto de un dominio, la especificación formal de un producto exigiría definir todos sus requisitos. Sin embargo, dentro de un dominio, la especificación para cada producto de los requisitos comunes y con un valor fijo es superflua y puede omitirse. Este ahorro posibilita que un DSL pueda ser más abstracto que cualquier **lenguaje de propósito general** (**GPL**, *General Purpose Language*).

[3] El diccionario de la Real Academia de la Lengua Española incluye las siguientes definiciones para "analogía":

"Relación de semejanza entre cosas distintas."

"Razonamiento basado en la existencia de atributos semejantes en seres o cosas diferentes."

"Creación de nuevas formas lingüísticas, o modificación de las existentes, a semejanza de otras; por ejemplo, los pretéritos tuve, estuve, anduve se formaron por analogía con hube."

que satisface los requisitos comunes de todos los productos. Después, se van incorporando capas de flexibilización que implementan los requisitos variables[4].

- Los requisitos comunes de una familia de productos suelen ser más estables que los requisitos variables. EDD separa la implementación de los requisitos comunes (el ejemplar) de la implementación de los requisitos variables (los módulos que flexibilizan el ejemplar).

- Como señalan M. Morisio et al. [MRS02], la mayoría de las líneas de productos de éxito provienen de software legado. La obtención de una familia de productos por analogía con un ejemplar previamente desarrollado sigue esta evolución natural.

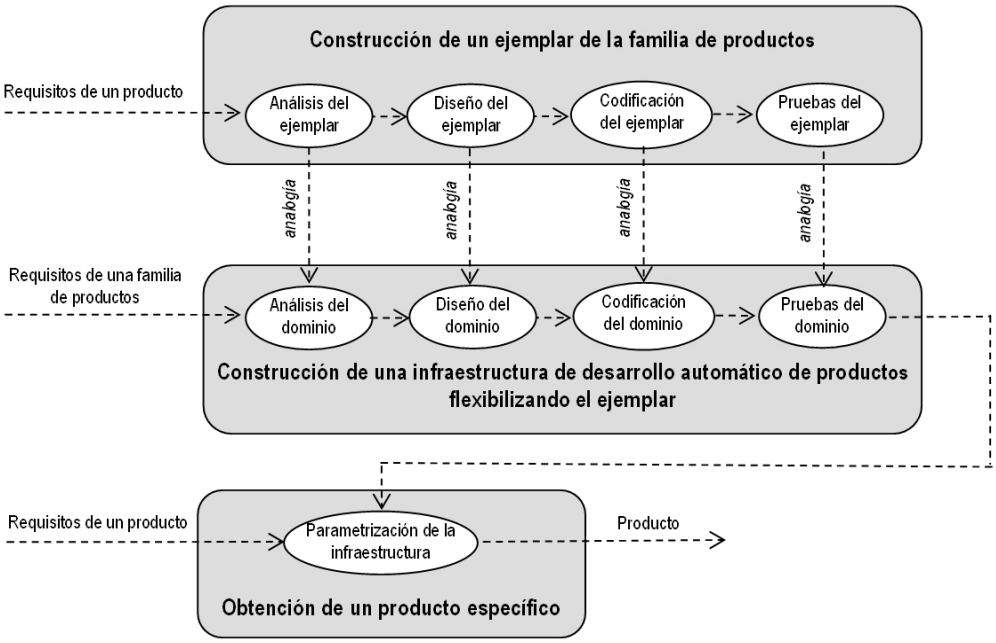


Figura 1.2. Proceso EDD de desarrollo de familias de productos.

1.4. Lenguaje de transformaciones ETL

Para definir formalmente la analogía entre el ejemplar y los demás productos de una familia, y posibilitar así la generación automática de estos últimos, es conveniente contar con algún mecanismo de flexibilización que cumpla las siguientes propiedades:

[4] En la sección 3.2.4 se explicará la integración de EDD con el modelo de ciclo de vida en espiral propuesto por B. Boehm [Boe88].

- **Modularidad**. La flexibilización debería poder descomponerse en módulos con la máxima cohesión y el mínimo acoplamiento.

- **No invasividad**[5]. Una flexibilización es invasiva si exige la manipulación previa del ejemplar para introducirle "etiquetas" que hagan referencia a los módulos que implementan los requisitos variables (figura 1.3). La invasión del ejemplar tiene los siguientes inconvenientes:

  - Posibilidad de introducir accidentalmente cambios no deseados en el ejemplar.

  - Deslocalización de los errores del "código variable". Un error en la gestión de un requisito variable puede encontrarse, además de en el módulo correspondiente, en la "etiqueta" del ejemplar que referencia al módulo.

  - Introducción de acoplamiento del ejemplar hacia los "módulos variables". Una modificación en la implementación de los requisitos variables podrá implicar cambios en el ejemplar.

  En una flexibilización no invasiva, el ejemplar se mantendrá intacto y cada módulo se responsabilizará de la gestión íntegra de los requisitos variables (qué parte del ejemplar debe modificarse y en cómo debe realizarse la modificación).

- **Aplicable a cualquier tipo de producto software**. Como señala I. Sommerville [Som05, página 5], "el software no son sólo los productos, sino todos los documentos asociados y la configuración de datos que se necesitan para hacer que estos productos operen de manera correcta".

- **Capaz de conseguir productos finales eficientes**.

- **Con algún medio para detectar automáticamente errores en la flexibilización**.

En la sección 2.1 se verá que el código es el producto del ciclo de vida con mayor tasa de reutilización. Por esta razón, las técnicas que comúnmente se emplean para generalizar código y posibilitar su reutilización parecen buenas candidatas para la flexibilización de un ejemplar. En el capítulo 5 se examinará la viabilidad de muchas de

[5] La no invasividad entre módulos (*invasiveness*) es una de las metas de la orientación a aspectos [Lad03a].

estas técnicas para la flexibilización y se comprobará que padecen serias limitaciones. Con el fin de superar dichas limitaciones se propone el lenguaje **ETL** (*Exemplar Transformation Language*), original de esta tesis.

ETL facilita la descomposición de las flexibilizaciones en módulos que actuarán transversalmente[6] y de forma no invasiva sobre el ejemplar. Dispone de operaciones para combinar los módulos y de primitivas para expresar modificaciones sobre el ejemplar.

Esta tesis incluye una implementación de ETL en Ruby [Rub07] que permite la flexibilización de ejemplares escritos en cualquier lenguaje. Concretamente, los capítulos 4, 5 y 6 mostrarán ejemplos de flexibilización de programas escritos en Java, C++ y TRANSACT SQL; de juegos de prueba escritos en Java y Modula-2; y de documentación escrita en HTML y Javadoc.

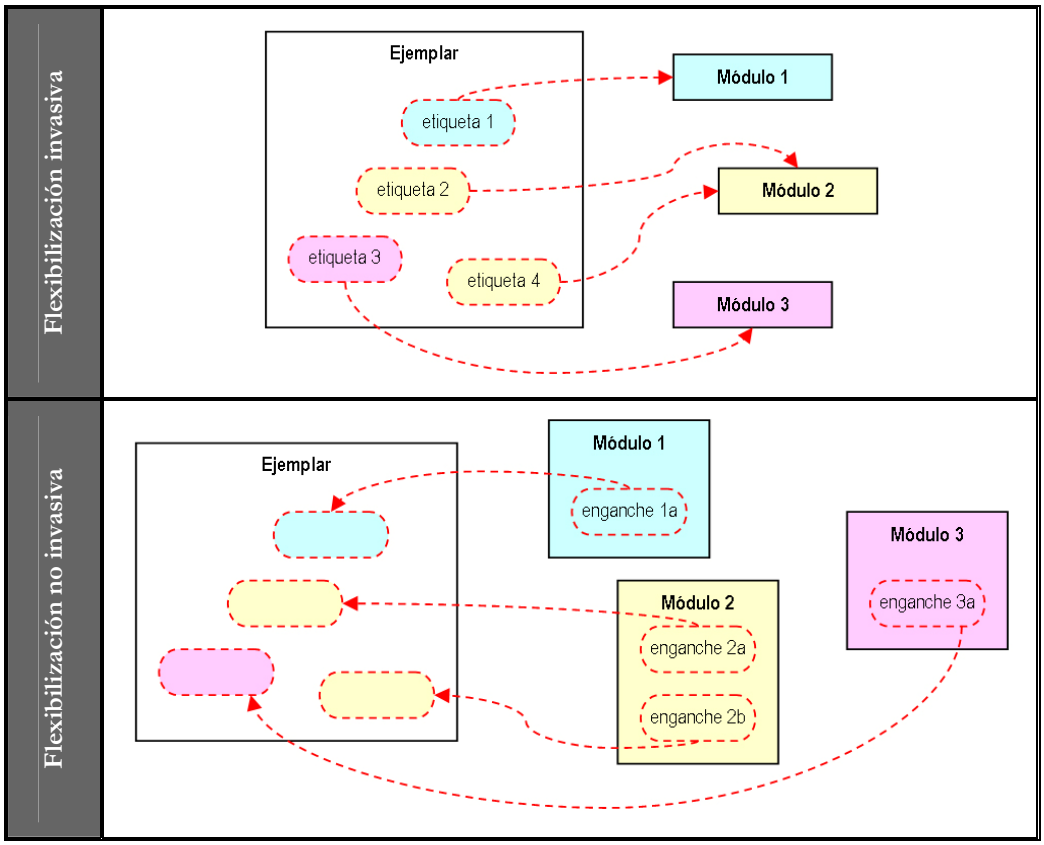


Figura 1.3. Flexibilizaciones invasivas y no invasivas de un ejemplar.

[6] Un módulo ETL puede actuar sobre distintas partes de un ejemplar. Por ejemplo, si el ejemplar fuese un programa orientado a objetos, un módulo ETL podría actuar sobre distintas clases.

1.5. Organización de la tesis

El resto de la tesis se organiza del siguiente modo:

- El **capítulo 2** es un estado del arte que resume:

    - Los logros, problemas y áreas de investigación en la reutilización del análisis, el diseño, la codificación y las pruebas de software.

    - Tres recientes e importantes propuestas para el desarrollo de familias de productos: la programación generativa, las factorías de software y el desarrollo dirigido por modelos.

    - Las principales aproximaciones para construir compiladores de DSLs.

- El **capítulo 3** expone el proceso de desarrollo EDD.

- El **capítulo 4** presenta la especificación del lenguaje de transformaciones ETL y resume su implementación con el lenguaje Ruby.

- El **capítulo 5** muestra, a través de dos ejemplos, distintas maneras de construir una familia de productos. Concretamente, se compara EDD con la metodología de programación generativa propuesta por la metodología K. Czarnecki y U. Eisenecker en [CE00], y se examina cómo aplicar las técnicas comunes de generalización de código a la flexibilización de un ejemplar y cuáles son sus limitaciones.

- **El capítulo 6** incluye diversos supuestos prácticos que ilustran la potencia de EDD y ETL.

- El **capítulo 7** resume las conclusiones de esta tesis y los trabajos futuros que pueden desarrollarse a raíz de ella.

Por último, adjunto al texto de la tesis se incluye un CDROM que contiene la implementación de ETL y el código de los ejemplos resueltos en los capítulos 4, 5 y 6.

2

Estado del arte

Though some pundits have suggested that there has been more reuse of the word "reuse" than practice of it, it's undoubtedly the case that a major area of progress in our industry has involved enabling reuse.

S. Mellor, K. Scott, A. Uhl, D. Weise, *MDA Distilled.*

Como se argumentó en el capítulo anterior, el desarrollo de familias de productos constituye un paso decisivo hacia la reutilización sistemática de software y la obtención de economía de alcance.

La **sección 2.1** del presente capítulo resume importantes logros, problemas y áreas de investigación en la reutilización del análisis, el diseño, la codificación y las pruebas de software.

Los obstáculos para la reutilización son múltiples. Un repositorio de componentes reutilizables debe satisfacer la mayor demanda posible asegurando, a la vez, la fácil localización de sus componentes. La búsqueda en un repositorio se simplifica si está convenientemente organizado o consta de pocos componentes. Normalmente, para maximizar la oferta de un repositorio minimizando la cantidad de sus componentes, se persigue la generalidad y ortogonalidad de los componentes. De manera que, parametrizándolos, satisfagan necesidades particulares y, combinándolos, produzcan nuevos componentes. La parametrización de un componente se facilita si su grado de ocultación es de "caja negra". Así, se evita la necesidad de conocer los detalles de

implementación de un componente para parametrizarlo. La ortogonalidad de los componentes aumenta al incrementar la cohesión de cada componente y disminuir el acoplamiento entre componentes [YC79]. Por otro lado, para que se pueda automatizar la reutilización de componentes, es imprescindible que estén especificados formalmente. Como se verá en las **secciones 2.1.1, 2.1.2, 2.1.3 y 2.1.4**, a medida que crece la abstracción de los componentes, mayor es la dificultad de especificarlos formalmente con sencillez y concisión y, por tanto, menor es la automatización de su reutilización. Por ejemplo, la parametrización y combinación automática de componentes de código como macros, funciones, clases… está muy difundida[7], mientras que la reutilización automática de diseño es muy inferior y, la de análisis, aún menor.

La **sección 2.2** presenta tres recientes e importantes propuestas para el desarrollo de familias de productos: la programación generativa, las factorías de software y el desarrollo dirigido por modelos. La infraestructura de desarrollo automático que persiguen estas propuestas es un compilador capaz de traducir especificaciones DSL de alto nivel de abstracción en productos finales. La **sección 2.3** resume las principales aproximaciones para construir compiladores de este tipo.

2.1. La reutilización de software

2.1.1. La reutilización en el análisis

El principal objeto de reutilización en la fase de análisis son las especificaciones de requisitos [BFP94, Jus96]. Habitualmente, las especificaciones son informales y se expresan en lenguaje natural [CR00], lo que dificulta su parametrización y disminuye su capacidad para combinarse y formar nuevas especificaciones.

Para paliar estas carencias, existen algunas propuestas de uso de distintos grados de formalización:

- El **modelado semiformal**, cuyo representante más difundido es UML (*Unified Modeling Language*) [BRJ99], suele ser expresivo y relativamente conciso. Sin embargo, las posibilidades que ofrece para la parametrización son bastante limitadas. A título de ejemplo, el diagrama de clases UML de la figura 2.1 representa una pila genérica (*Stack*)

[7] No así su búsqueda y selección, para las que existe un escaso soporte automático debido a que normalmente la semántica de los componentes de código se especifica en lenguaje natural.

y dos contenedores alternativos para almacenar sus elementos (*Vector* y *Deque*). Los parámetros se expresan con trazo discontinuo.

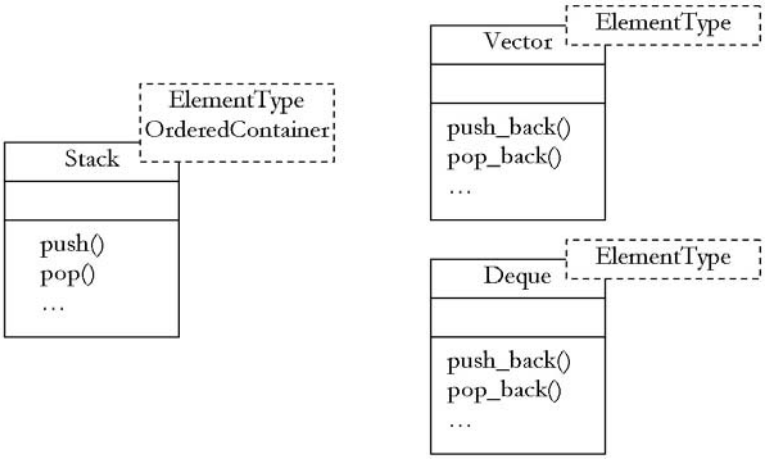


Figura 2.1. Pila genérica y dos contenedores alternativos para almacenar sus elementos.

K. Czarnecki [CE00, página 208] critica la capacidad de parametrización de los diagramas de clases de UML por las siguientes razones:

1. No permiten expresar restricciones sobre el tipo de un parámetro.

2. No permiten expresar los métodos que obligatoriamente deberá tener una clase utilizada como parámetro real.

3. Un parámetro se puede usar con diversos fines, por ejemplo, para expresar el tipo de una superclase (herencia parametrizada), el tipo de un objeto asociado, el tipo del argumento de un método… En UML es muy difícil, si no imposible, representar explícitamente el propósito de un parámetro.

Con el lenguaje de restricciones OCL (*Object Constraint Language*) [WK03] se puede mejorar la precisión de los diagramas UML. Sin embargo, algunos autores [Fow03, Lar02] desaconsejan su uso al considerarlo excesivamente costoso.

– Los **lenguajes de especificación formal,** como Z [Dil94] ó TUG (*Tree Unified with Grammar*) [Chi03], están libres de ambigüedad y permiten la definición de especificaciones genéricas que pueden particularizarse y combinarse. Sin embargo, como indica M. Chechik [Che98], el uso de los métodos formales está poco extendido porque suelen considerarse costosos y difíciles de usar[8].

---

[8] *"There is a strong resistance against adopting formal methods in practice, especially outside the domain of safety-critical systems. The primary reason for this resistance is the perception of software developers regarding the applicability of formal methods – these*

Por otro lado, exceptuando los requisitos de seguridad, de rendimiento y, en general, los requisitos no funcionales, con frecuencia los requisitos recuperados de un repositorio sólo son parcialmente reutilizables y necesitan algún tipo de adaptación [LGL02].

Estos problemas explican el escaso nivel de reutilización logrado en la fase de análisis [LMV97, MKKW99].

Muchas investigaciones asumen la naturaleza informal de los requisitos y se orientan a la mejora de la organización y las búsquedas en repositorios de especificaciones escritas en lenguaje natural [CR00, GI93] o en algún lenguaje de modelado semiformal [RW04].

Para que los requisitos puedan reutilizarse independientemente, interesa que sean autocontenidos. Lamentablemente, esta situación no es frecuente. Si, por ejemplo, tres requisitos X, Y, Z hacen referencia a un cuarto requisito R, pueden tomarse varias decisiones:

- Incluir R en cualquier especificación que reutilice a X, Y ó Z. De esta forma, X, Y, Z pasan a ser autocontenidos. Como inconveniente, la especificación pierde concisión e incluso puede ser redundante (si incluye a X e Y, contendrá dos veces a R).

- Para la reutilización de X, Y, Z se impone la reutilización explícita de R.

MRAM (*Method for Requirements Authoring and Management*) [MKKW99] es una metodología que aborda la representación y gestión de las dependencias entre requisitos de forma arbórea. Los nodos de un árbol pueden ser requisitos o discriminantes de tres tipos: exclusión mutua (*Single Adaptor*), lista de alternativas (*Multiple Adaptor*) o rama opcional (*Option*). Por ejemplo, en la figura 2.2 el requisito B no tiene discriminantes. Si se reutiliza B, también se incluyen A, C, D y E. En la figura 2.3 se representa una exclusión mutua. Si se selecciona G, se descarta la otra rama, la cual permite la reutilización de H, I, J o cualquiera de sus combinaciones. Por último, la figura 2.4 representa la reutilización opcional del requisito K.

---

methods are considered to be hard (require a level of mathematical sophistication beyond that possessed by many software developers), expensive, and not relevant for 'real' software systems".

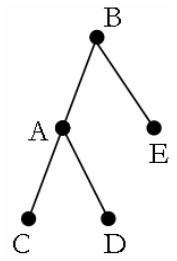


Figura 2.2

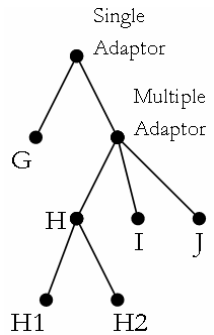


Figura 2.3

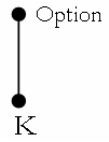


Figura 2.4

Diagramas MRAM para la representación de dependencias entre requisitos

2.1.2. La reutilización en el diseño

2.1.2.1. Diseño arquitectónico

Los patrones o estilos [SG96] arquitectónicos son soluciones de eficacia probada a problemas que aparecen con frecuencia en el diseño arquitectónico[9]. El libro "*Pattern-Oriented Software Architecture. A System of Patterns.*" [BMRSS96], conocido popularmente como POSA1[10], contribuyó de manera decisiva a la difusión de este tipo de patrones. En él, se enumeran los siguientes beneficios derivados del uso de patrones:

- Permiten la reutilización de soluciones arquitectónicas de calidad.
- Son de gran ayuda para controlar la complejidad de un diseño.
- Facilitan la documentación de diseños arquitectónicos.
- Proporcionan un vocabulario común que mejora la comunicación entre diseñadores.

---

[9] *"A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their reponsibilities and relationships, and the ways in which they collaborate".* [BMRSS96]

[10] Posteriormente, se publicaron dos secuelas de este libro conocidas como POSA2 (Schmidt, D.; Stal, M.; Rohnert, H.; Buschmann, F. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects.* John Wiley & Sons, 2000) y POSA3 (Kircher, M.; Jain, P. *Pattern-Oriented Software Architecture, Volume 3, Patterns for Resource.* John Wiley & Sons, 2004).

POSA1 ofrece un catálogo de patrones arquitectónicos organizados, según su propósito, en cuatro categorías: *From Mud to Structure, Distributed Systems, Interactive Systems* y *Adaptative Systems* (figura 2.5).

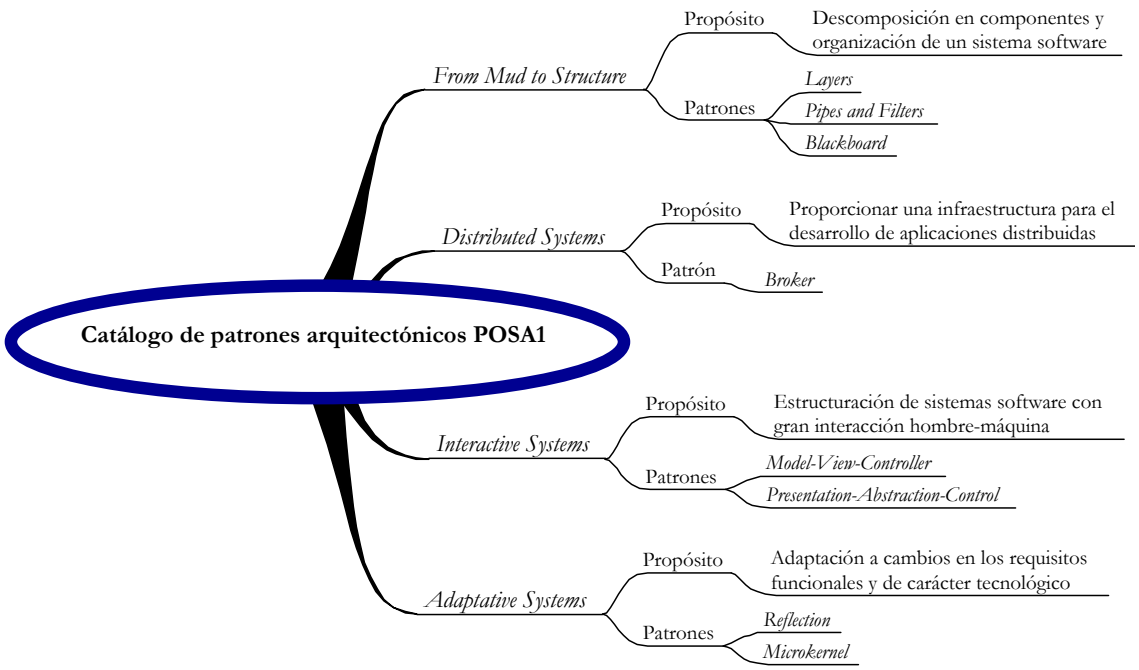


Figura 2.5. Catálogo de patrones arquitectónicos incluido en POSA1.

En POSA1, los patrones se formulan informalmente según los siguientes apartados: nombre del patrón, sinónimos, ejemplo de aplicación, contexto o situaciones en las que el patrón puede utilizarse, propósito o tipo de problema que se resuelve, estrategia utilizada en el planteamiento de la solución, aspectos estructurales de la solución, comportamiento dinámico del patrón, implementación posible, variantes, usos conocidos, relación con otros patrones, beneficios y desventajas.

Posteriormente, se han publicado nuevos catálogos donde los patrones también se registran de manera informal [Fow02, IBM06, TMQH+03, TRHM+04]. Ante la proliferación de patrones arquitectónicos, con el fin de automatizar la elección de los patrones más adecuados a problemas concretos, se han desarrollado sistemas de ayuda a la decisión [SF04] e intentos de formalización del propósito de los patrones [Sch01] y de cómo su uso afecta a la calidad de una aplicación informática [KK99, ZBJ04].

Reconocida la utilidad de los patrones arquitectónicos para la documentación y el mantenimiento de aplicaciones informáticas, se han producido investigaciones en ingeniería inversa para la extracción de patrones en código de lenguajes de programación [PG02].

2.1.2.2. Patrones de diseño

Análogos a los patrones arquitectónicos, los patrones de diseño son buenas soluciones de problemas que aparecen de forma recurrente en el diseño detallado. El catálogo "*Design Patterns: Elements of Reusable Object-Oriented Software*" [GHJV94], cuyo germen fue la tesis doctoral de E. Gamma [Gam91], marcó un hito en la reutilización del diseño. Antes de su publicación, aparecieron otros catálogos de patrones menos abstractos, más vinculados a la implementación [Cop91, Gla90]. La extraordinaria difusión de [GHJV94] ha llevado a que el término genérico "patrón de diseño", que debería englobar a cualquier tipo de patrón (arquitectónico, de diseño detallado…), se utilice por defecto para hacer referencia al tipo de patrón orientado a objetos que recoge el citado catálogo.

[GHJV94] contiene 23 patrones organizados según su propósito en tres categorías: de creación, estructurales y de comportamiento (figura 2.6)

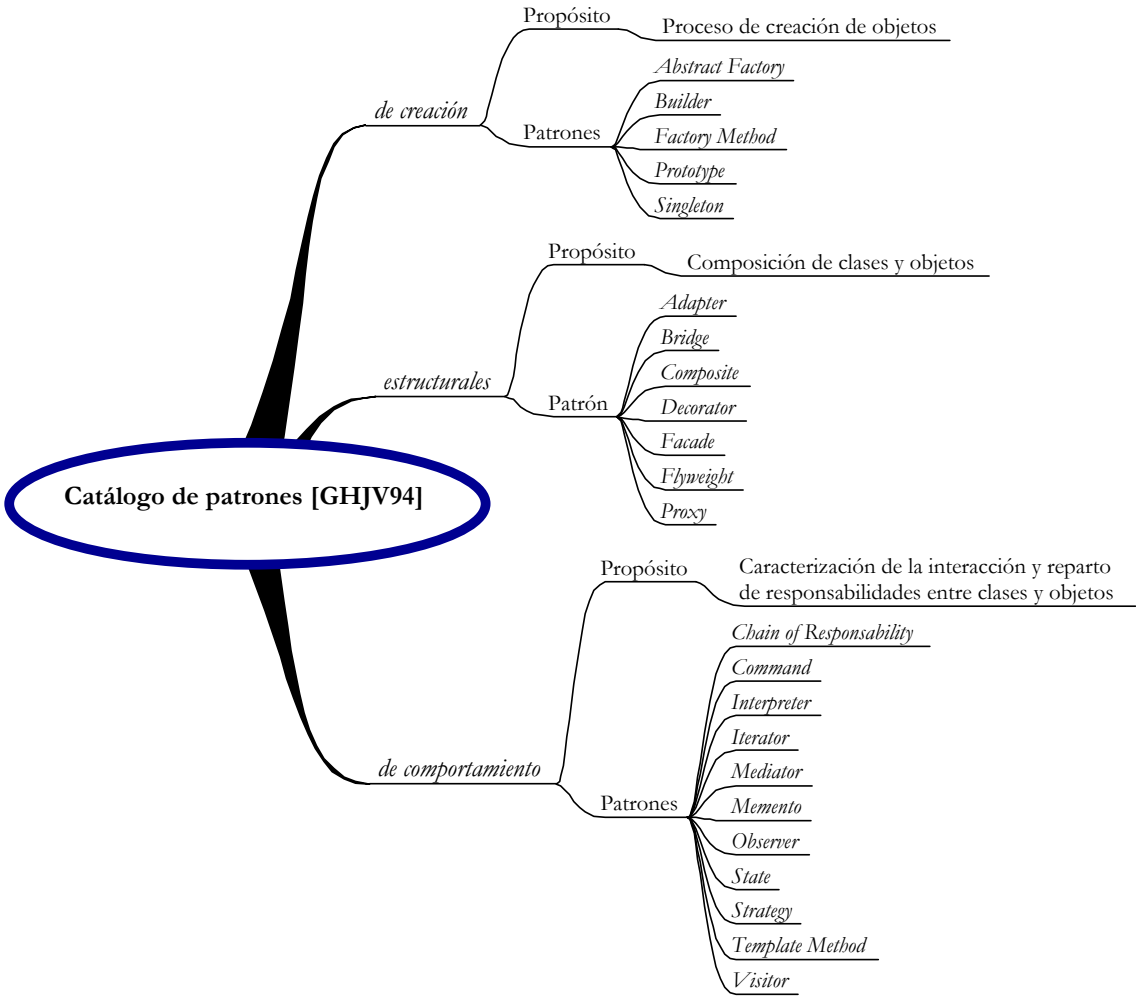


Figura 2.6. Catálogo de patrones de diseño incluido en [GHJV94].

En [GHJV94], los patrones se especifican con la notación de modelado semiformal OMT (*Object Modeling Technique*) (diagramas de clases y diagramas de objetos para representar los aspectos estructurales, diagramas de interacción para expresar el comportamiento dinámico) acompañada de una descripción en lenguaje natural organizada según el nombre del patrón, propósito, sinónimos, motivación, aplicabilidad, participantes en los diagramas OMT, consecuencias de la utilización del patrón, consejos de implementación, código de ejemplo escrito en C++ y Smalltalk, usos conocidos y patrones relacionados.

Posteriormente, se han publicado numerosos catálogos al estilo [GHJV94], por ejemplo, la serie "*Pattern Languages of Program Design*", que va por el quinto volumen [CS95, VCK96, MRB97, HFR99, MVN06] y "*The Pattern Almanac 2000"* [Ris00], que contiene alrededor de 700 patrones organizados en 70 categorías.

Los patrones de diseño suscitan diversos problemas y controversias. A continuación, se presentan algunas áreas de investigación en torno a ellos.

Especificación

La ambigüedad con que están expresados los patrones puede provocar su mal uso y limita su gestión automática mediante herramientas informáticas.

Existen algunas propuestas de formalización basadas en la lógica, como el BPSL (*Balanced Pattern Specification Language*) [TL03], donde los aspectos estructurales y dinámicos se especifican combinando la lógica de primer orden (FOL, *First Order Logic*) con acciones de lógica temporal (TLA, *Temporal Logic Actions*).

Otros autores recomiendan UML complementado con Acciones Semánticas [MCL04] u OCL [FKGS04] para eliminar ambigüedades.

Ocultación

Tal como se especifican en [GHJV94], los patrones son unidades de reutilización de caja blanca [GS04]: para usar el patrón no basta con conocer su propósito y "parámetros variables", sino que es imprescindible sumergirse en sus interioridades y adaptarlo manualmente a cada situación particular.

Algunos lenguajes de programación, como Ruby [Rub07], disponen de la capacidad de extensión suficiente para encapsular ciertos patrones de diseño (ofreciendo una interfaz que sólo expone los parámetros variables y ocultando la codificación de la parte permanente del patrón y de la gestión de la parte variable). A modo de ejemplo, en [TH01] aparecen encapsulados los patrones *visitor, delegate, observer* y *singleton*.

Cuando no es posible encapsular un patrón con un lenguaje de programación de propósito general, se puede utilizar una estrategia generativa. Se elabora un lenguaje específico de dominio[11] (DSL, *Domain Specific Language*), cercano al usuario, para el ajuste de los parámetros variables y se genera automáticamente el código ejecutable correspondiente. S. MacDonald et al. [MSSA+02] han desarrollado dos herramientas (figura 2.7) que siguen este enfoque:

- CO_2P_2S (*Correct Object-Oriented Pattern-based Programming System*) ofrece una representación gráfica abstracta de diversos patrones almacenados previamente. El usuario configura de forma visual un patrón especificando exclusivamente sus parámetros y la herramienta aprovecha la utilidad *Javadoc* [Jav06] para generar código ejecutable a partir de plantillas prefabricadas.

- Meta-CO_2P_2S posibilita la implementación de nuevos patrones (representación gráfica parametrizable y generación de código ejecutable).

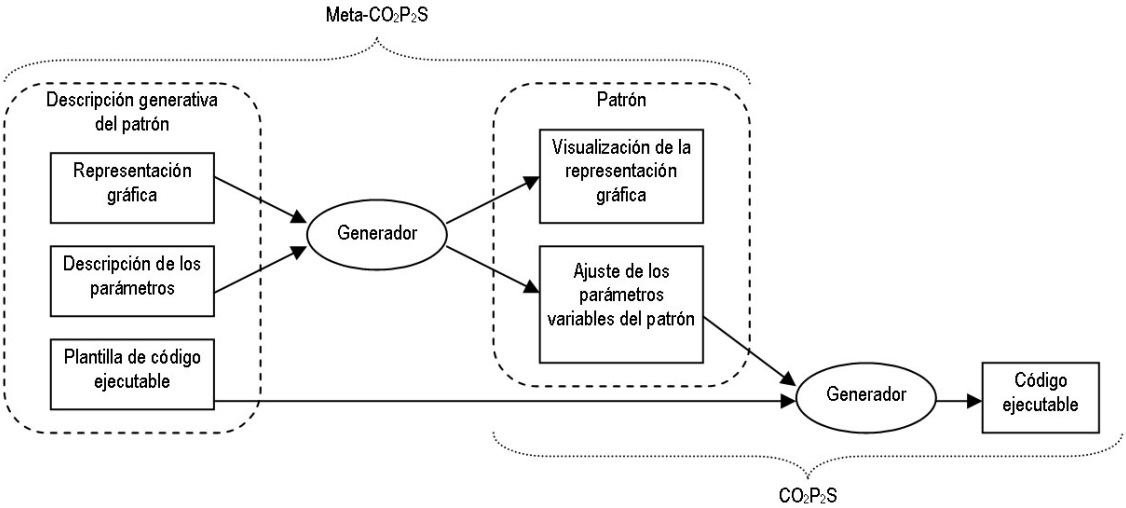


Figura 2.7. Modelo generativo ofrecido por CO_2P_2S y Meta-CO_2P_2S para la encapsulación de patrones de diseño.

[11] Una estrategia común para elevar la abstracción de un lenguaje es delegar decisiones en el compilador o intérprete. El número de decisiones que puede delegar un lenguaje de propósito general (GPL*, General Purpose Language*) sin mermar su generalidad es muy reducido, lo que limita las posibilidades de abstracción. Un DSL se adecua a un ámbito particular donde pueden darse por sentado ciertas tareas y conceptos que se manejan con frecuencia, lo que contribuye a incrementar significativamente su nivel de abstracción y cercanía al usuario.

Acoplamiento

Como señalaron E. Yourdon y L. L. Constantine en 1979 [YC79], el objetivo fundamental de cualquier diseño es conseguir un sistema mantenible. Sólo en casos excepcionales se sacrificará este objetivo para lograr una mayor velocidad de proceso o un menor tamaño de código. El logro de este objetivo pasa por alcanzar el mínimo acoplamiento entre sus componentes y la máxima cohesión de cada uno de ellos.

Sin embargo, el arquitecto C. Alexander [AISJ+77], considerado de forma unánime como el padre del diseño basado en patrones, enunció lo siguiente:

"Es posible hacer edificios enlazando patrones, de un modo poco preciso. Un edificio construido así es una mezcolanza de patrones. No es denso. No es profundo. También es posible juntar patrones de modo que muchos patrones se solapen en un mismo espacio físico: el edificio es muy denso; tiene muchos significados representados en un espacio reducido; y a través de esa densidad, se hace profundo."

Muchos autores [GHJV94, YXA00a] interpretan esta postura afirmando que un diseño orientado a objetos que se limite a encadenar patrones (*Stringing patterns*) degenerará en una superpoblación de pequeñas clases triviales y redundantes. La tendencia opuesta, la imbricación profunda (*Overlapping patterns*), conlleva una alto acoplamiento. Por ejemplo, la figura 2.8 muestra un diseño incluido en [MSU98]. La clase *Translator* participa en dos patrones (*Template Method* y *Builder*) desempeñando roles distintos. Un cambio en cualquiera de los patrones que afecte a *Translator*, se propagará al otro patrón. En [MB01] puede encontrase un estudio empírico sobre el nivel de acoplamiento en diseños con uso intensivo de patrones y su impacto en la calidad final.

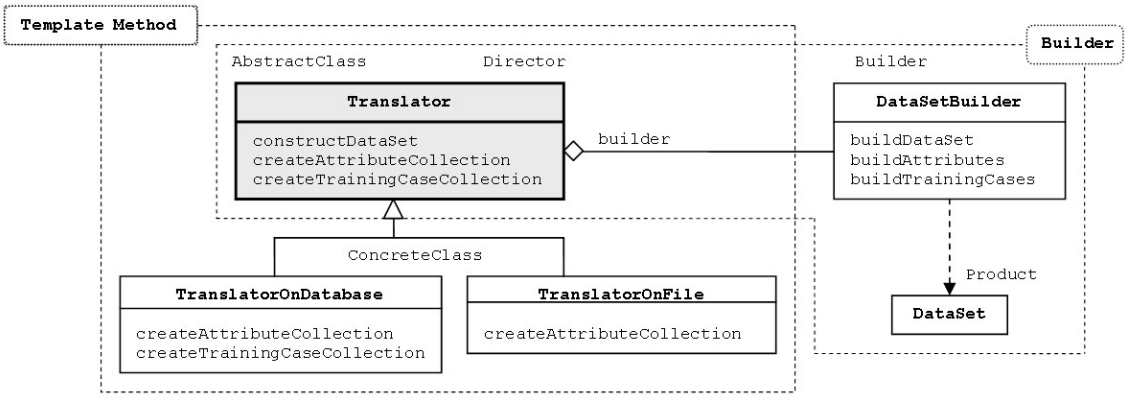


Figura 2.8. Ejemplo de diseño con solapamiento de patrones (tomado de [MSU98]).

Numerosas investigaciones se orientan hacia la composición eficaz de patrones de diseño. Por ejemplo, S. M. Yacoub y H. H. Ammar proponen la metodología POAD (*Pattern-Oriented Analysis and Design*) [YA03] que, apoyándose en una definición de interfaz

válida para lo que llaman "patrones construccionales"[12], guía la reutilización sistemática de patrones y su composición manteniendo el control sobre el acoplamiento. Además, para dar soporte a la metodología, han desarrollado la herramienta POD (*Pattern-Oriented Design tool*) [YXA00b].

Mantenimiento de software y patrones de diseño

Existe consenso sobre la contribución de los patrones de diseño a la mejora de la documentación del software [ACHL02, KSRP99, PK98] y, de esta manera, a su mantenimiento. Sin embargo, muchos patrones incorporan cierto grado de complejidad para aumentar la adaptabilidad del software [BJY01]. Aunque algunos experimentos justifican esta adición [PUTB+01], la mayoría de los estudios empíricos no permiten establecer una correlación entre el uso de patrones y una disminución de los cambios necesarios durante el mantenimiento [BSWM+03, Vok04].

Admitiendo la utilidad de los patrones de diseño para la ingeniería inversa, se han producido numerosas investigaciones sobre la extracción de patrones en código de lenguajes de programación orientada a objetos, como C++ [BF03] o Eiffel [WT05]. Inicialmente, los métodos de extracción eran manuales [SMB96]. Posteriormente, se han empleado métodos que transforman el código en representaciones intermedias [CLDG+05, ZLB04] que después se contrastan automáticamente con librerías de patrones. Dado el parecido estructural entre algunos de los patrones más populares (*Composite* vs *Decorator*, *State* vs *Strategy, Bridge* vs *Adapter*…) [FBFL05], algunas investigaciones van más allá de una comparativa estructural e incorporan el análisis de aspectos dinámicos [HML03].

Eficiencia

Muchos de los patrones recogidos en [GHJV94] ofrecen la adaptabilidad de los diseños por medio de enlace dinámico (*dynamic binding*), lo que merma la eficiencia en tiempo de ejecución de sistemas que implementen un número considerable de estos patrones.

---

[12] "*A constructional design pattern is an object-oriented design pattern, it has well defined interfaces, and it provides a solution that is an abstraction of a common design structure in the form of a class model. The name constructional is given to these patterns because they are used in constructing the structure of the application design. Constructional design patterns are design components that can be glued together at a high design level. This composition defines the application overall solution structure. Constructional design patterns do not necessarily map to structural design patterns defined by Gamma et al. Gamma's behavioral patterns, such as State or Observer patterns, can also be considered constructional because they provide class diagrams as a structure for composition. Patterns that do not provide a solution structure as class models are not considered constructional; e.g. the Pipes-and-Filters or Layered-architecture patterns*".

Algunos autores [Alex01, CE00] han propuesto diseños alternativos para estos patrones sustituyendo el enlace dinámico (en tiempo de ejecución) por el uso de genericidad (en tiempo de compilación).

2.1.2.3. Marcos de trabajo

Generalmente, la reutilización de código se consigue incorporando componentes prefabricados (clases, subprogramas…) almacenados en librerías. El criterio que cohesiona los componentes de una librería es su aptitud para resolver cierto tipo de problemas. Es decir, su adecuación para un dominio específico.

Cuando existe una gran proximidad entre las aplicaciones informáticas de un dominio, se puede ir más allá y tratar de reutilizar el esqueleto común de las aplicaciones, técnicamente denominado "marco de trabajo" [13,14]. Un marco de trabajo representa las decisiones de diseño que son comunes a su dominio. Así, los marcos hacen hincapié en la reutilización del diseño frente a la reutilización del código.

El uso de marcos de trabajo no sólo acelera la construcción de aplicaciones, además, hace que las aplicaciones tengan estructuras parecidas, por lo que son más fáciles de mantener y resultan más consistentes para los usuarios [GHJV94].

Existen numerosos estudios experimentales que avalan la capacidad de los marcos de trabajo para mejorar la productividad en el desarrollo de software y la calidad de las aplicaciones finales [Mat99, MN96, MRS02].

El desarrollo basado en marcos de trabajo consta fundamentalmente de dos procesos [MRS02]:

1. **La construcción del marco de trabajo.** Para el éxito de este proceso, debe determinarse con precisión el dominio. Las técnicas de análisis de dominio [CHW98] facilitan la detección de los aspectos comunes de todas las aplicaciones del dominio, que se implementarán en el marco, y de los puntos de variación [Sri99]), que serán parametrizados por los usuarios del marco. Acotado el dominio, se pasa a construir el

---

[13] *"A framework defines an aplication skeleton that can be refined and customised by developers to create a range of different applications"* [KRW02].

[14] "*A framework is a set of cooperating classes that make up a reusable design for a specific class of software"* [Deu89].

marco que, como señalan M. Morisio et al. [MRS02], suele obtenerse generalizando software legado[15].

2. **La construcción de aplicaciones particularizando el marco de trabajo.** Los marcos de trabajo no son abstracciones de caja negra[16]. Esto significa que el usuario deberá comprender la implementación del marco para particularizarlo, lo que implica una curva de aprendizaje considerable [SLB00]. Siguiendo la terminología de [CE00], mientras que los marcos de trabajo se ubican en el "espacio de la solución", los usuarios pertenecen al "espacio del problema" y manejan puntos de variación abstractos propios de este espacio. Como la encapsulación del marco no es de caja negra, los usuarios se ven obligados a aventurarse en el espacio del problema y traducir manualmente sus puntos de variación conceptuales (*mapping problem* [KRW02]). Como inconveniente adicional, se liga la especificación de los programas a la implementación del marco. Si se altera dicha implementación, las especificaciones pasarán a ser inservibles.

Colisiones entre marcos de trabajo

Una aplicación compleja puede reutilizar varios marcos, lo que abre la posibilidad de solapamientos y colisiones [MB97]. Por ejemplo, en la reutilización basada en marcos de trabajo, respecto a la basada en librerías de componentes, se da una inversión del flujo de control. El código nuevo de una aplicación no invoca a elementos prefabricados, sino que ocurre al revés, el marco de trabajo es quien invoca al nuevo código[17]. La figura 2.9 representa una aplicación que reutiliza los marcos A, B y C. El posible acceso simultáneo de los marcos al código de la aplicación que los particulariza plantea los problemas clásicos de la concurrencia (estados inconsistentes, interbloqueos…).

---

[15] *"Successful frameworks have evolved from reengineering long-lived legacy applications, abstracting from them the knowledge of principal software designers"*

[16] Según [GS04], los marcos de trabajo son abstracciones de caja gris (*gray box abstractions*)

[17] Popularmente, esto se conoce como "el principio de Hollywood" (*Hollywood principle: "Don't call us – we call you"*)

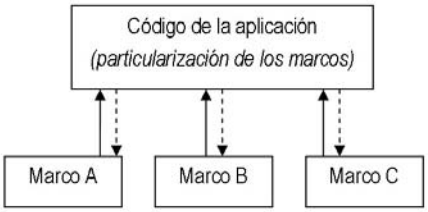


Figura 2.9. Inversión del flujo de control y colisiones en los marcos de trabajo.

2.1.3. La reutilización en la codificación

Quizás, la forma más primitiva de reutilización de código sea la expansión de texto basada en **macros**. Lamentablemente, esta técnica impone un fuerte acoplamiento entre la macro y el contexto desde el que se expande, pues su interacción se realiza mediante variables con el mismo identificador.

Las **funciones**[18] que disponen de transparencia referencial, es decir, que transforman un mismo conjunto de entradas en el mismo conjunto de salidas independientemente del contexto desde el que se invocan, soslayan el problema de acoplamiento de las macros. Además, la reutilización intensiva de funciones lleva a organizarlas en librerías, aumentándose el grano de la reutilización.

En ocasiones, conviene que las funciones "tengan memoria" y puedan consultar resultados de invocaciones previas o información utilizada anteriormente por otras funciones. En los años 60 y 70 proliferaron librerías con funciones que compartían información mediante alguna estructura global de datos. Desgraciadamente, esta organización padece un acoplamiento análogo al que sufren las macros, ya que cualquier cambio en la estructura de datos se propaga a todas las funciones que la acceden.

Los **tipos abstractos de datos** [Wir76] y, posteriormente, la **orientación a objetos** [Mey00] superan el problema anterior encapsulando la estructura de datos junto con las funciones que la manipulan. Mientras se mantenga la interfaz de las funciones del tipo abstracto, los cambios en la estructura de datos sólo afectan a la implementación de dichas funciones, deteniéndose la propagación de las modificaciones.

Dada la gran envergadura de los programas, a mediados de los 90 surgió la tendencia de agrupar las clases en una entidad de grano mayor denominada **componente software** [Szy02].

---

[18] El término "subprograma" engloba a las "funciones" y a los "procedimientos". Generalmente, los lenguajes orientados a objetos utilizan el término "método" en lugar de subprograma.

Como puede verse, el carácter formal del código ha posibilitado su reutilización efectiva desde los albores de la informática.

En la siguiente sección, se resume una modalidad de reutilización de código novedosa que ha inspirado parte de esta tesis, la **orientación a aspectos.**

2.1.3.1. Programación orientada a aspectos

Dada la capacidad limitada del cerebro humano para procesar información[19], una buena estrategia para manejar cuestiones complejas es dividirlas en otras más simples que se tratan por separado. Este principio, conocido como "separación de las preocupaciones" (SOC, *Separation Of Concerns*), ha sido defendido desde antiguo por grandes investigadores como D. L. Parnas [Par72] o E. W. Dijkstra [Dij76].

La mayoría de la metodologías de análisis y de diseño, así como la práctica totalidad de los lenguajes de programación, proponen construcciones (subprogramas, clases…) para organizar un sistema informático en unidades modulares. Aunque dichas construcciones facilitan la creación y encapsulación de las cuestiones funcionales que constituyen el núcleo de una aplicación, son insuficientes para capturar otro tipo de cuestiones que, por esta razón, suelen estar dispersas por toda la aplicación. Las cuestiones del primer tipo se denominan "centrales" (*core concern*), mientras que las del segundo tipo se llaman "transversales" (*crosscutting concerns)* o aspectos. Ejemplos de aspectos [Lad03a] son la interacción entre componentes, la persistencia, la sincronización, los históricos de ejecución (*logging*), estrategias para el uso eficiente de la memoria…

La falta de modularización de los aspectos puede percibirse cuando existe:

1. Código enredado (*code tangling*): algunos módulos implementan más de un concepto (figura 2.10).

2. Código disperso (*code scattering*): un concepto aparece implementado en más de un módulo (figura 2.11).

---

[19] Generalmente, se acepta que la memoria humana se divide en tres sistemas: memoria sensorial, memoria operativa o a corto plazo y memoria a largo plazo. La memoria operativa es el sistema donde el usuario maneja la información a partir de la cual está interactuando con el medio ambiente. Según G. A. Miller [Mil56], esta información está limitada aproximadamente a 7 ± 2 conceptos durante 20 segundos si no se repasa.

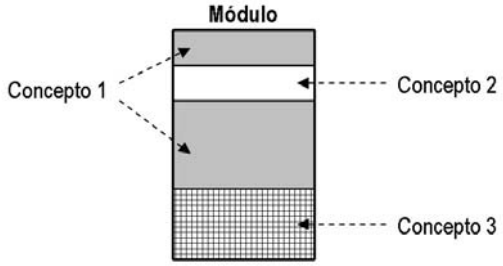


Figura 2.10. Código enredado

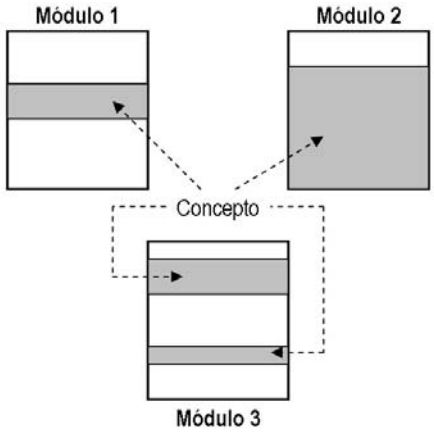


Figura 2.11. Código disperso

La encapsulación de los aspectos produce el aumento de la cohesión modular y la disminución del acoplamiento entre módulos, cualidades que redundan en una mejora de la productividad (desarrollo en paralelo), de la reutilización de módulos y del mantenimiento (localización de los cambios y control de su propagación).

El desarrollo orientado a aspectos se apoya en dos herramientas fundamentales: el lenguaje que facilita la implementación de los aspectos y el tejedor (*weaver*) que, para producir el sistema final, mezcla los aspectos con los conceptos centrales (figura 2.12).

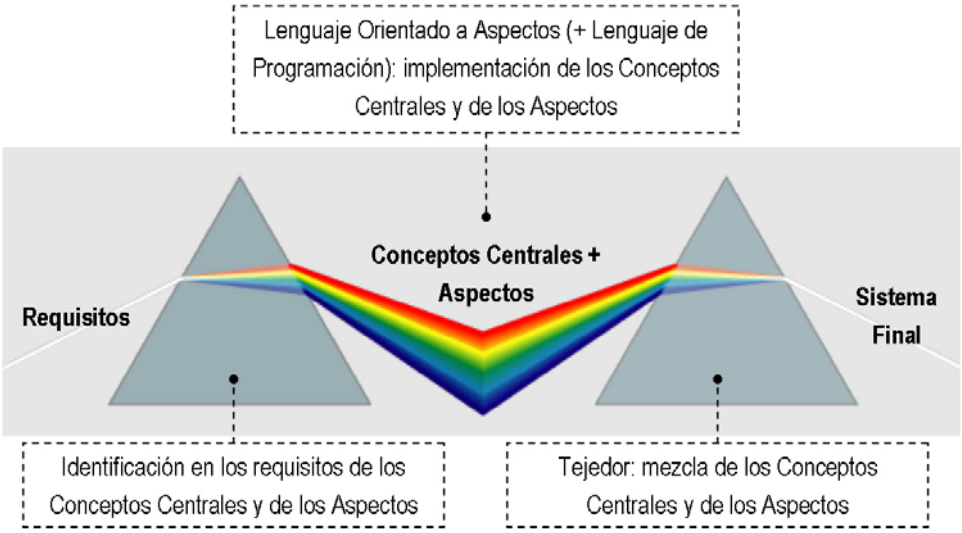


Figura 2.12. Resumen del proceso de desarrollo orientado a aspectos.

Para que el tejedor pueda realizar su trabajo automáticamente, además de la expresión del comportamiento de los aspectos, el lenguaje de implementación debe facilitar la descripción de la combinación entre aspectos y conceptos centrales. En general, los lenguajes orientados a aspectos se plantean como extensiones de lenguajes de programación. El comportamiento de los aspectos se describe con el lenguaje subyacente (Java en el caso de AspectJ [KHHK+01, Lad03a] e Hyper/J [OT01], C++ en el caso de

AspectC++ [SLU05], C# en el caso de AspectC# [Kim02]...). La parte novedosa de un lenguaje orientado a aspectos es la expresión de la combinación entre aspectos y conceptos centrales, cuyas cualidades deseables son [CE00]:

1. Mínimo acoplamiento entre aspectos y conceptos centrales.

2. Capacidad de enlace (*binding time*) estático y dinámico entre aspectos y conceptos centrales.

3. Adición no invasiva de los aspectos. Los aspectos incorporan cualidades transversales a los conceptos centrales. Conviene que esta adición no implique la adaptación manual de los conceptos centrales. Cuando un arquitecto aborda la construcción de un sistema informático, se enfrenta al dilema del subdiseño frente al sobrediseño. Al comienzo de un proyecto, desconoce todos los "requisitos definitivos". Si opta por subdiseñar, el diseño inicial puede verse gravemente afectado ante un cambio en los requisitos. Por otro lado, si opta por sobrediseñar, corre el riesgo de complicar innecesariamente el diseño[20] para anticiparse a modificaciones que nunca se producirán. Frente a los procesos formales, como el Proceso Unificado de Rational [Kru00], que están orientados a la construcción de sistemas de gran tamaño, se sitúan los procesos ágiles, como eXtreme Programming [Beck99] o Scrum [RJ00], que buscan la reacción efectiva frente a cambios en proyectos más reducidos. Los procesos ágiles defienden el subdiseño frente al sobrediseño[21], es decir, creen que la construcción de sistemas debe ser pragmática y ceñirse a los requisitos "en tiempo presente". R. Laddad [Lad03a] opina que la orientación a aspectos está en sintonía con esta tendencia y que la adición no invasiva de aspectos puede ser una herramienta muy valiosa para adaptar un sistema evitando su modificación manual.

AspectJ

A continuación, se utilizará el lenguaje AspectJ para ilustrar los principales elementos de un lenguaje de aspectos.

En AspectJ, la superposición de los aspectos sobre los conceptos centrales (o sobre otros aspectos) se expresa mediante tres elementos:

---

[20] *design bloat*

[21] Principio acuñado popularmente como *YAGNI*: *"You Aren't Gonna Need It"*.

1. *Join points*. Son los lugares de un programa donde puede engancharse un aspecto. AspectJ ofrece un abanico muy amplio de tipos de *join points*: llamada a métodos, acceso a atributos, inicialización de clases y objetos… Algunos *join points* se consideran conflictivos y no están disponibles para el programador[22].

2. *Pointcuts*. Seleccionan uno o varios *join points*.

3. *Advices*. Cuando el flujo de ejecución de un programa alcanza un *pointcut*, se produce un salto a su *advice* asociado, que contiene el código Java que implementa parte o la totalidad del comportamiento del aspecto. Este funcionamiento es muy similar al de la Programación Dirigida por Eventos, donde el trozo de código del programa original delimitado por un *pointcut* "hace de botón" que dispara un evento cuando el flujo de ejecución lo alcanza y el *advice* "hace de manejador del evento". Los *advices* se pueden ejecutar antes (*before*), después (*after*) o sobre (*around*) la activación del *pointcut*. Algunos *join points* tienen asociado un contexto accesible desde el *advice* correspondiente. Por ejemplo, el contexto de la llamada a un método contiene los objetos llamante y llamado.

Si se solapan los *pointcuts* de varios aspectos pueden producirse colisiones. Para resolver esta clase de conflictos, AspectJ permite fijar un orden de prioridad entre aspectos mediante la sentencia *declare precedence*.

Líneas de investigación

Numerosos investigadores apoyan la orientación a aspectos [Lad03b, VV00, WBM99], sin embargo, otros la critican [Ale03] pues consideran, por ejemplo, que la verificación de un programa orientado a aspectos se dificulta porque a) la introducción de aspectos complica el hilo de ejecución de los programas, y por tanto, la realización de pruebas de caja transparente; b) el comportamiento de un aspecto puede depender del contexto del programa sobre el que se combine, lo que impide la prueba aislada del aspecto.

Algunas investigaciones [YLM04] buscan una metodología que facilite la identificación sistemática de aspectos en los requisitos. Otras, orientadas hacia la ingeniería inversa, persiguen la detección automática de aspectos en código legado [Bre05].

Quizás, el campo de investigación más activo sea el desarrollo de nuevos lenguajes de aspectos y tejedores que reúnan las cualidades deseables de combinación entre conceptos

[22] Los *join points* disponibles para el programador se denominan *exposed join points*.

centrales y aspectos mencionadas anteriormente [GR04, KHHK+01, Kim02, OT01, TPSG+02].

2.1.4. La reutilización en las pruebas

El auge de los procesos ágiles [Bec99, RJ00], que atribuyen una enorme importancia a la prueba del software, ha contribuido de forma decisiva al desarrollo y la difusión de marcos de trabajo para la automatización de las pruebas de unidades: JUnit para Java [JUn06], NUnit para C# [NUn06], CppUnit para C++ [Cpp06], RubyUnit para Ruby [RUn06], HttpUnit para aplicaciones web [HUn06]… Lamentablemente, como se indicó anteriormente, los marcos de trabajo no son abstracciones de caja negra. Existen algunas propuestas de aumento de la abstracción de marcos para pruebas de unidades mediante un enfoque generativo [HC05].

La formalización de las pruebas de unidades constituye un paso decisivo para su reutilización. Varias investigaciones respaldan que la reutilización de componentes posibilita, en gran medida, la reutilización de las pruebas de dichos componentes [Edw01, Lon98, Mic97]. Algunos autores consideran que la reutilización de pruebas puede incrementarse notablemente en dominios específicos, aprovechando la cercanía entre las aplicaciones de una familia [DS05, MMWO94].

2.2. Desarrollo de familias de productos

2.2.1. La programación generativa

La programación generativa (*GP, Generative Programming*) es un paradigma de ingeniería del software basado en el desarrollo de familias de sistemas[23]. El producto final de la GP es un modelo generativo capaz de sintetizar todos los programas[24] de una familia a partir de

---

[23] *"Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge."* [CE00]

[24] El resultado de un modelo generativo no tiene porque ser un programa completo. Puede ser un programa parcial, un componente de otros programas [JS00, WLKL04] o cualquier artefacto software: documentación, juegos de pruebas… En esta tesis, se seguirá la tendencia más difundida de utilizar el término "familia de programas" para designar de forma general familias de todo tipo de artefactos software.

especificaciones de alto nivel de abstracción. La figura 2.13 representa los componentes básicos de un modelo generativo, que son:

- **El espacio del problema**. Es el ámbito de actuación del ingeniero de aplicaciones. Consiste en uno o varios DSLs que facilitan la especificación abstracta de programas.

- **El espacio de la solución**. Es el conjunto de marcos de trabajo y componentes que se reutilizan en la generación de los programas objeto.

- **El espacio de configuración**. Es el compilador que convierte las especificaciones DSL en programas objeto.

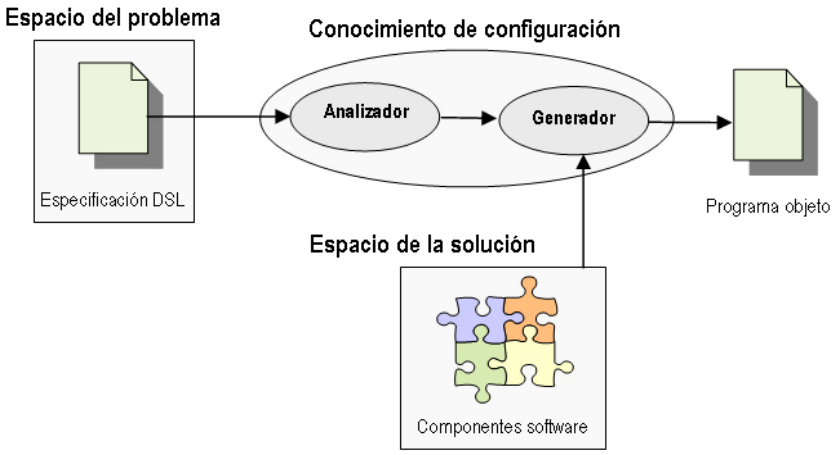


Figura 2.13. Arquitectura propuesta en [CE00] para modelos generativos.

Apoyándose en la ingeniería de dominio [DE06], K. Czarnecki y U. Eisenecker [CE00] proponen un **proceso de desarrollo** de modelos generativos organizado en las siguientes etapas:

1. Análisis del dominio del modelo.

    1.1. Determinación del alcance del dominio.
    1.2. Análisis de los aspectos comunes y variables del dominio. Modelado del dominio.

2. Diseño del dominio.

    2.1. Diseño de una arquitectura común para los programas de la familia. Identificación de los componentes del espacio de la solución.
    2.2. Diseño de los DSLs del espacio del problema.
    2.3. Diseño del conocimiento de configuración.

3. Implementación del dominio.

3.1. Implementación de los componentes del espacio de la solución.
3.2. Implementación de los DSLs del espacio del problema.
3.3. Implementación del conocimiento de configuración.

Los autores recomiendan aplicar este proceso de forma incremental e iterativa. Por ejemplo, el conocimiento de configuración se debería automatizar gradualmente: antes de utilizar generadores, los componentes del espacio de la solución se parametrizarían y ensamblarían manualmente.

El análisis del dominio de un modelo generativo es una actividad crítica que debe evitar:

- La omisión de puntos de variación relevantes, que dificultaría el mantenimiento del modelo.

- La inclusión de puntos de variación innecesarios, que complicarían e incrementarían el coste del modelo.

Existen varias metodologías para analizar dominios: FAST (*Family-Oriented Abstraction, Specification and Trans*lation) [WL99], ODM (*Organization Domain Model*ng) [ODM06], FODA (*Feature-Oriented Domain Analysis*) [KCHN+90]…

El modelado de características (Feature modeling) está integrado en la metodología FODA y ha sido aplicado con éxito en el desarrollo de sistemas de telecomunicaciones [GFA98, LKL02], librerías de plantillas [CE00], protocolos de red [BB02], sistemas embebidos [CBUE02]…

El elemento fundamental de un modelo de características es un diagrama arbóreo cuya raíz representa un concepto que tiene como descendientes sus características asociadas. Una característica es una propiedad relevante para el cliente. Suele utilizarse para detectar los puntos de variación de una familia de programas.

Existen diversas notaciones para los diagramas de características (figura 2.14). En esta tesis, se seguirá la más reciente [CHU04] (última columna de la figura 2.14), que utiliza la idea de cardinalidad para salvar las dificultades de representación que padecen otras notaciones [CBUE02].

Los diagramas de características ofrecen una representación de los puntos de variación de una familia de programas clara e independiente de su implementación [CE00]. Por ejemplo, el diagrama de la figura 2.15 indica que una persona en una determinada organización puede ser un empleado, un cliente, un accionista o cualquiera de las

combinaciones posibles ({empleado, cliente}, {empleado, accionista}, {cliente, accionista}, {empleado, cliente, accionista}). La figura 2.16 es el diagrama de clases UML equivalente, más confuso y dependiente de la implementación.

Notación FODA inicial [LKL02]	**Notación FODA extendida [Cza98, CE00]**	**Notación con cardinalidad [CHU04]**
Características obligatorias y opcionales	Características obligatorias y opcionales	Características obligatorias y opcionales
Características alternativas	Grupo or exclusivo	Grupo con cardinalidad <1-1>
No disponible	Grupo or inclusivo	Grupo con cardinalidad <1-k>
No disponible	Grupo or exclusivo con características opcionales	Grupo con cardinalidad <0-1>

Figura 2.14. Notaciones para diagramas de características.

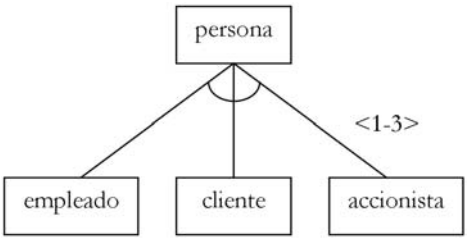


Figura 2.15. Diagrama de características "roles en una organización".

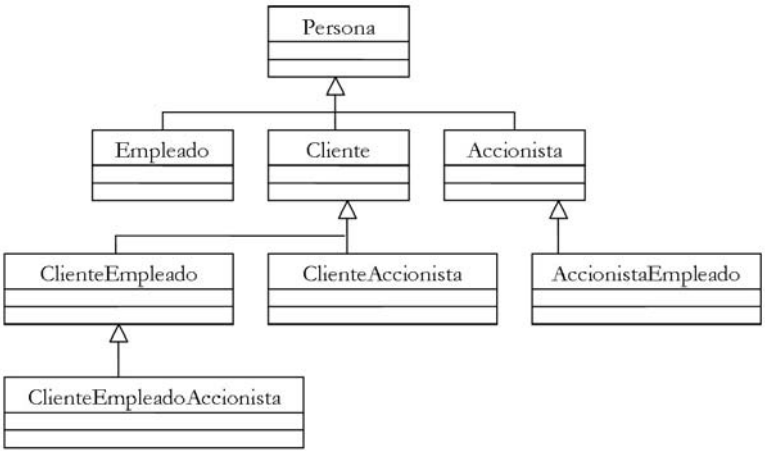


Figura 2.16. Diagrama de clases UML "roles en una organización.

2.2.2. Las factorías de software

Las factorías son fábricas o complejos industriales que producen artículos en masa y suelen caracterizarse por disminuir los gastos de transporte centralizando la producción, descomponer el proceso de fabricación en tareas elementales estandarizadas, disponer de mecanismos de control para asegurar la calidad de los productos, contar con trabajadores muy especializados y aplicar un alto grado de automatización del trabajo y de reutilización de artefactos [Cus89].

A finales de los 60, R. W. Bemer [Bem68] utilizó el término "factoría de software" para enfatizar la necesidad de estandarización y control en la fabricación de software. M. D. McIlroy [McI68] también usó este término para resaltar la conveniencia de la división modular de los programas y la reutilización de software.

La primera compañía que adoptó el término factoría de software para identificar una forma de organización del trabajo fue *Hitachi* en 1969 con su *Hitachi Software Works*. Posteriormente, se sumaron otras compañías como la norteamericana *System Development Corporation* en 1975 y las japonesas *NEC*, *Toshiba* y *Fujitsu* durante 1976 y 1977.

En la actualidad, *Microsoft* ha retomado el término factoría de software para nombrar una metodología propia de desarrollo de líneas de productos[25] [SF06]. En lo que sigue de esta tesis, se utilizará el término factoría con este significado.

---

[25] Según K. Czarnecki [CE00] y P. Clements [CN99], los términos línea de productos y familia de programas hacen referencia a dos formas diferentes de percibir un dominio. Desde un punto de vista orientado al problema, una línea de productos se define como un conjunto de programas que satisfacen alguna necesidad

Los elementos fundamentales de una factoría de software son:

1. **El esquema de la factoría**. Es un documento que recopila de forma estructurada:

   - Los productos que debe producir la línea (archivos de código fuente, documentos XML, modelos, ficheros de configuración, archivos SQL, juegos de prueba…)

   - Los elementos constituyentes de los productos (componentes software, patrones de diseño, marcos de trabajo, DSLs, herramientas y lenguajes para desarrollar los distintos artefactos…)

   - Procesos sobre cómo desarrollar, configurar y reutilizar los elementos constituyentes y cómo combinarlos para crear productos.

   Siguiendo el estándar IEEE 1471 [IEEE1471], un esquema se descompone según diferentes puntos de vista. Los esquemas suelen representarse de dos maneras:

   - Tablas donde las columnas indican conceptos y las filas niveles de abstracción. Por ejemplo, la figura 2.18 es el esquema propuesto por J. Zachman [Zac04] para aplicaciones empresariales. Cada celda es un punto de vista que aglutina productos, elementos constituyentes y procesos.

   - Grafos donde los nodos representan puntos de vista y los arcos relaciones entre puntos de vista. La figura 2.17 es un ejemplo de esquema incluido en [GS04]. Las relaciones sirven para mantener la coherencia entre los puntos de vista y capturar el conocimiento sobre como obtener los artefactos de un punto de vista a partir de los artefactos incluidos en otro punto de vista. Si las descripciones de los artefactos son formales, es posible automatizar la conversión, que puede ser completa o parcial. En ocasiones, la conversión entre puntos de vista no es única y su optimización depende del contexto. Por ejemplo, existen numerosas maneras de modelar la herencia para el almacenamiento de objetos en una base de datos relacional (utilizar una tabla para guardar una jerarquía completa; por cada clase hija, usar una tabla que incluya los atributos de la clase padre…) [Amb00]. Cada alternativa tiene sus

---

del mercado. Desde una perspectiva orientada a la solución, una familia de programas es un conjunto de programas software muy similares entre sí. En esta tesis, se utilizarán indistintamente los dos términos.

ventajas e inconvenientes, y su elección depende de cada situación particular. En estos casos, es muy interesante que la conversión sea parametrizable [GS04].

2. **La plantilla de la factoría**. Es una implementación de un esquema. Algunos autores [Fow05, GS04] creen que se avecina una auténtica revolución en los entornos de desarrollo integrados (IDE, *Integrated Development Environment*[26]). Los nuevos IDEs ofrecerán una gran extensibilidad y facilitarán la rápida implementación de DSLs. La propuesta de Microsoft en este sentido es la nueva versión de Visual Studio [VSt05]. Algunos IDEs (Intentional Programming [IS06, Sim99], Meta Programming System [Jet06]) tratarán de superar el tradicional aspecto textual del código permitiendo que el usuario trabaje directamente con árboles sintácticos o, preferiblemente, con diferentes vistas de los árboles. La plantilla de una factoría configurará los IDEs para adecuarlos a la línea de producción.

Los procesos fundamentales para la construcción de una factoría de software son el desarrollo de una línea de productos y el desarrollo de un producto concreto de una línea. La figura 2.19 resume los procesos y productos de una factoría. Puede encontrase una explicación más detallada y un ejemplo de la metodología en los capítulos 11 y 16 de [GS04] respectivamente.

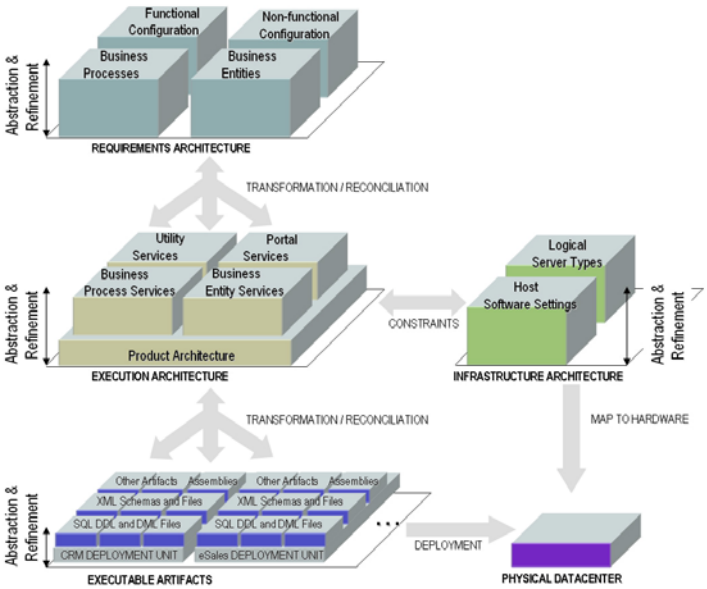


Figura 2.17. Representación mediante un grafo del esquema de una factoría de software propuesto por J. Greenfield [GS04]

[26] Como indica el *Free Online Dictionary on Computing* (http://foldoc.org/), en ocasiones el acrónimo IDE también se traduce por *Interactive Development Environment*.

	WHAT DATA	HOW FUNCTION	WHERE NETWORK	WHO PEOPLE	WHEN TIME	WHY MOTIVATION
SCOPE {contextual} Planner	List of Things Important to the Business Entity = Class of Business Thing	List of Processes the Business Performs Process = Class of Business Process	List of Locations in Which the Business Operates Node = Major Business Location	List of Organizations Important to the Business People = Major Organizational Unit	List of Events/Cycles Significant to the Business Time = Major Business Event/Cycle	Lists of Business Goals/Strategies Ends/Means = Major Business Goal/Strategy
BUSINESS MODEL {conceptual} Owner	e.g., Semantic Model Entity = Business Entity Relationship = Business Relationship	e.g., Business Process Model Process = Business Process I/O = Business Resources	e.g., Business Logistics System Node = Business Location Link = Business Linkage	e.g., Work Flow Model People = Organization Unit Work = Work Product	e.g., Master Schedule Time = Business Event Cycle = Business Cycle	e.g., Business Plan End = Business Objective Means = Business Strategy
SYSTEM MODEL {logical} Designer	e.g., Logical Data Model Entity = Data Entity Relationship = Data Relationship	e.g., Application Architecture Process = Application Function I/O = User Views	e.g., Distributed System Architecture Node = I/S Function (Processor, Storage, etc.) Link = Line Characteristics	e.g., Human Interface Architecture People = Role Work = Deliverable	e.g., Processing Structure Time = System Event Cycle = Processing Cycle	e.g., Business Rule Model End = Structural Assertion Means = Action Assertion
TECHNOLOGY MODEL {physical} Builder	e.g., Physical Data Model Entity = Segment/Table/etc. Relationship = Pointer/Key/etc.	e.g., System Design Process = Computer Function I/O = Data Elements/Sets	e.g., Technology Architecture Node = Hdw/System Software Link = Line Specifications	e.g., Presentation Architecture People = User Work = Screen Formats	e.g., Control Structure Time = Execute Cycle = Component Cycle	e.g., Rule Design End = Condition Means = Action
DETAILED REPRESENTATIONS {out-of-context} Subcontractor	e.g., Data Definition Entity = Field Relationship = Address	e.g., Program Process = Language Statement I/O = Control Block	e.g., Network Architecture Node = Address Link = Protocol	e.g., Security Architecture People = Identity Work = Job	e.g., Timing Definition Time = Interrupt Cycle = Machine Cycle	e.g., Rule Specification End = Sub-condition Means = Step
FUNCTIONING ENTERPRISE	e.g.: DATA	e.g.: FUNCTION	e.g.: NETWORK	e.g.: ORGANIZATION	e.g.: SCHEDULE	e.g.: STRATEGY

Figura 2.18. Representación tabular del esquema de una factoría de software propuesto por J. Zachman [Zac04]

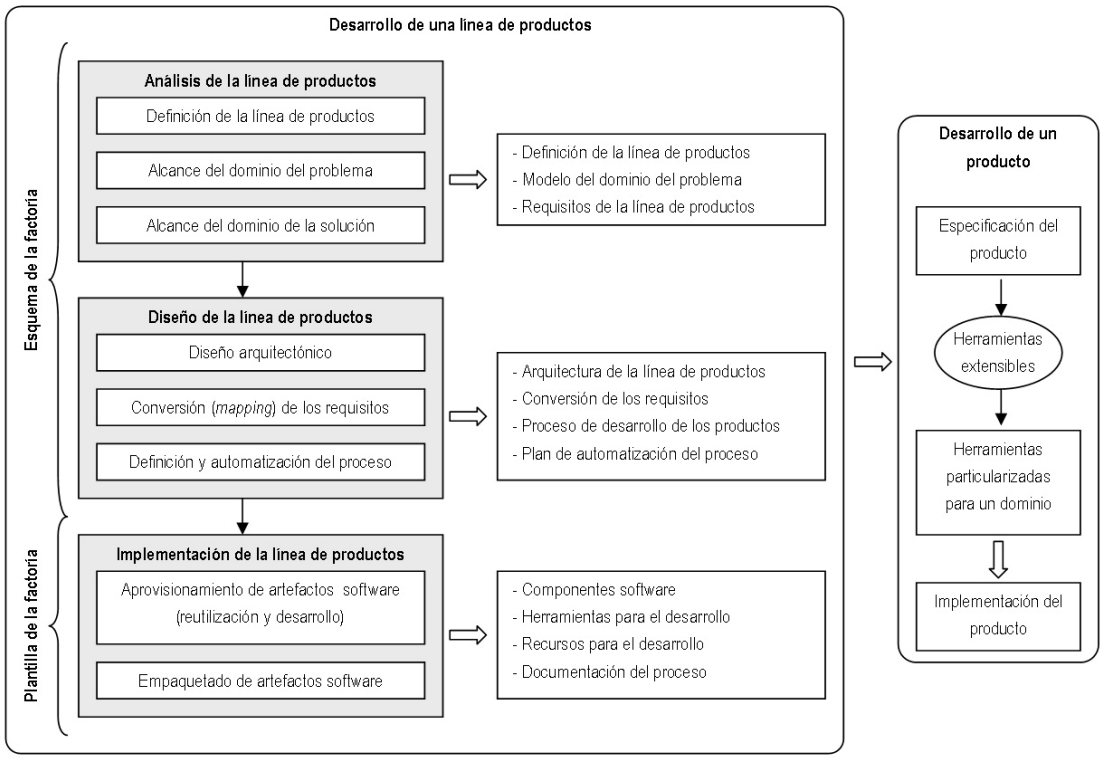


Figura 2.19. Principales procesos y productos de una factoría de software.

2.2.3. El desarrollo dirigido por modelos

Con el fin de centrar la presentación de esta sección, se citan algunas definiciones del término "modelo":

Un modelo es una simplificación de la realidad. Un modelo proporciona los planos de un sistema. Los modelos pueden involucrar planos detallados, así como planos más generales que ofrecen una visión global del sistema en consideración. Un buen modelo incluye aquellos elementos que tienen una gran influencia y omite aquellos elementos menores que no son relevantes para el nivel de abstracción dado.

[BRJ99, página 5]

Un modelo de un sistema es una descripción o especificación de dicho sistema y su entorno con algún propósito en particular. Con frecuencia, se representa combinando gráficos y textos. Estos últimos, pueden escribirse en lenguaje natural o en algún lenguaje de modelado.

[MDA03, página 2-2]

Un modelo es un conjunto de elementos que describe una realidad física, abstracta o hipotética. Un buen modelo sirve como medio de comunicación; es más barato de construir que un sistema real; y sirve de soporte para la implementación. El grado de detalle de un modelo puede variar desde un boceto gráfico hasta un modelo totalmente ejecutable.

[MSUW04, página 13]

Tradicionalmente, los modelos han servido como medio de comunicación entre clientes, analistas, diseñadores y programadores [CD94, Fow03]. Durante el análisis, los modelos se han utilizado para representar los conceptos de un dominio y para especificar las interfaces externas de un sistema. En el diseño, los modelos han servido para representar los planos que guían la implementación. Dada la variedad de personas, en cuanto a formación e intereses, que trabajan con modelos, los lenguajes de modelado, en general, han optado por sacrificar formalismos en aras de una mayor claridad. La primera versión de UML, el lenguaje de modelado más difundido en la actualidad, es un claro exponente de esta tendencia.

La utilización de modelos, ha recibido numerosas críticas, especialmente por los partidarios de los procesos de desarrollo ágiles [Bec99, RJ00]:

- Los modelos suelen ser poco comprensibles para el cliente. En la comunicación cliente-analista es preferible emplear prototipos.

- Los lenguajes de modelado tipo UML son ambiguos, lo que impide su traducción automática a código ejecutable y fuerza a los programadores a tomar decisiones que no aparecen reflejadas en ningún diagrama.

- En ocasiones, durante las pruebas y el mantenimiento se retoca el código sin que se actualicen los modelos. Como resultado, los modelos son una fuente de inconsistencia que no contribuye a mejorar la documentación.

- En definitiva, el producto verdaderamente relevante del proceso de desarrollo es el código. Los esfuerzos de desarrollo deben centrarse en construir y mantener un código de calidad fomentando el uso intensivo de pruebas, refactorizando el código [FBBO+99]…. Sólo es justificable el uso de modelos sencillos en casos puntuales.

En contraposición a estas críticas, el desarrollo dirigido por modelos (MDD, *Model Driven Development*) considera que los modelos son productos de primera categoría que permiten el avance hacia mayores niveles de abstracción y reutilización [MSUW04]. La arquitectura dirigida por modelos (MDA, *Model Driven Architecture*) [MDA03] del consorcio

OMG (*Object Management Group*) es la principal representante del MDD y persigue solventar algunos de los problemas fundamentales del uso de modelos, entre los que cabe destacar:

- La eliminación de la ambigüedad en los modelos, posibilitándose la traducción automática de estos a código ejecutable.

- La creación de un estándar que facilite la reutilización de modelos y su intercambio entre distintas herramientas de modelado.

Según el grado de abstracción, MDA establece tres puntos de vista para un sistema informático:

1. El **modelo independiente de computación** (CIM, *Computation Independent Model).* Representa el dominio de un sistema y sirve como medio de comunicación entre el cliente y el desarrollador.

2. El **modelo independiente de plataforma** (PIM, *Platform Independent Model).* Representa un sistema obviando su entorno de ejecución. Ejemplos de plataformas son Java, CORBA, .NET o los sistemas operativos Linux, Solaris, Microsoft…

3. El **modelo específico de plataforma** (PSM, *Platform Specific Model).* Representa un modelo incluyendo los detalles de su entorno de ejecución. Este modelo es directamente traducible a código ejecutable.

Además, MDA contempla la descomposición en modelos dentro de un mismo nivel de abstracción. Incluso reconoce la posibilidad de definir y combinar modelos transversales en el sentido de la orientación a aspectos [MSUW04, SSRG+05].

Dentro de los partidarios de MDA, existen dos corrientes [Hay04, McN03]: los *traduccionistas* [MB02], que abogan por modelos capaces de producir aplicaciones completas mediante un proceso de traducción automático, y los *elaboracionistas* [KWB03], que apuestan por la producción parcial de aplicaciones. Para unos y otros, la primera opción para escribir modelos es UML.

Las versiones iniciales de UML padecían serias carencias para representar el comportamiento y definir la semántica de los modelos. La versión 2.0 [UML20] supera muchas de estas carencias. Sin embargo, algunos autores [FGDS06] consideran que esta nueva versión tiene un grado de complejidad excesivo que dificulta su manejo.

Cuando UML es insuficiente, MDA propone dos posibilidades:

1. Extender UML desarrollando un nuevo **perfil** *(profile)* por medio del lenguaje de restricciones OCL y los **estereotipos** *(stereotypes)* de UML. Un ejemplo notorio de esta posibilidad es el perfil *Executable UML* [MB02], que permite la definición de modelos computables y es de propósito general. Otros autores [Sil03, TSS04, WZZ03], con el fin de simplificar los modelos, optan por definir perfiles computables para dominios específicos.

2. Construir o **metamodelar** un nuevo lenguaje de modelado, generalmente específico para un dominio [Pel02], utilizando el metalenguaje MOF (*Meta-Object Facility*) [MOF06] para describir su sintaxis abstracta. UML está descrito en MOF, que a su vez está descrito en el propio MOF [MSUW04]. Para facilitar la reutilización y el intercambio de modelos, se ha creado el estándar XMI (*XML Model Interchange*) [XMI05], que establece cómo derivar un esquema XML desde la sintaxis MOF de un lenguaje de modelado y cómo traducir los modelos escritos en ese lenguaje a documentos XML.

El traduccionista S. J. Mellor [MSUW04, capítulo 10] considera que MDA reconcilia los partidarios de UML con los partidarios de los procesos ágiles: "Si un modelo es ejecutable, es operacionalmente lo mismo que el código. Así, los principios ágiles para la construcción de código también son aplicables a la construcción de modelos ejecutables". Además, propone el siguiente **proceso de desarrollo** MDA [MSUW04, capítulos 11 y 12]:

1. Delimitación del dominio y posible descomposición jerárquica del mismo en subdominios.

2. Formalización del conocimiento del dominio (o de cada subdominio).
   - 2.1. Identificar los requisitos del dominio.
   - 2.2. Abstraer el conocimiento del dominio en algún grupo de conceptos.
   - 2.3. Seleccionar, o desarrollar si fuera necesario, un lenguaje de modelado apropiado para representar los conceptos.
   - 2.4. Expresar formalmente los conceptos en este lenguaje.
   - 2.5. Comprobar el modelo resultante del paso anterior.

3. Construir los puentes PIM-PSM
   - 3.1. Especificar las funciones de conversión
   - 3.2. Si es necesario, añadir marcas de forma no invasiva[27] al modelo.

[27] Sin modificar directamente el modelo.

3.3. Comprobar las funciones de conversión.
3.4. Examinar la completitud de los modelos.
3.5. Transformar los modelos

Como puede comprobarse, los pasos 1, 2.1, 2.2, 2.3, 3.1 y 3.3 sirven para el desarrollo de una línea de productos, mientras que los pasos 2.4, 2.5, 3.2, 3.4 y 3.5 se ocupan del desarrollo de un producto particular de la línea. Pueden encontrase otros procesos similares en [KMHC05] y [MA02].

Actualmente, existe una cantidad considerable de herramientas comerciales y de código abierto que soportan parcialmente MDA, por ejemplo, ArcStyler [AS06], OptimalJ [OJ06], AndroMDA [AM06], NetBeans Metadata Repository [NBMR06], ModFact [MF06], Eclipse Modeling Framework [EMF06], Rational Rose [RR06], EclipseUML [EU06], Poseidon [Pos06], ArgoUML [AU06], CodaGen Architect [CA06]…

En la sección 2.6.1 se resume un elemento clave de MDA: el lenguaje QVT de transformación de modelos.

Aclaración terminológica: ¿modelos o programas?

Los modelos y los programas son especificaciones de software. Como indica, J. Greenfield [GS04, páginas 226-229 y 318-319], la distinción entre "programa" y "modelo computable" es muy difusa. A continuación, se muestra la debilidad de algunos criterios utilizados habitualmente como discriminantes:

- La notación. Puede parecer que los lenguajes de programación tienen una notación textual mientras que los lenguajes de modelado tienen una notación gráfica. Sin embargo, existen lenguajes de programación gráficos, como el lenguaje que acompaña los *Lego Mindstorms* [Min06], y lenguajes de modelado formales textuales, como Z [Dil94], o semiformales con una fuerte componente textual, como el OCL de UML.

- El estilo de la especificación. Existen lenguajes de programación imperativos y lenguajes de programación declarativos (es más, los lenguajes de programación imperativos tienden a enriquecerse declarativamente y al revés). Históricamente, los lenguajes de modelado han sido de corte declarativo, sin embargo, los diagramas de transición de estados son un ejemplo de lenguaje imperativo.

- El propósito. Tradicionalmente, los modelos han servido como medio de comunicación entre personas (clientes, analistas, diseñadores…). Sin embargo, los

modelos computables son especificaciones formales también orientadas a la "comunicación con la máquina".

Siguiendo la sugerencia de J. Greenfield, en esta tesis no se distinguirá entre modelo computable, especificación formal y programa. Sin embargo, sí se diferenciará entre GPL y DSL.

2.2.4. Resumen comparativo

La GP, las factorías del software y el MDD persiguen la reutilización sistemática de software y el aumento de la abstracción de los lenguajes formales de desarrollo abordando la resolución colectiva de programas en dominios específicos y realizando un uso intensivo de generadores de código.

Los procesos de desarrollo incluidos en las secciones anteriores distinguen dos etapas fundamentales[28]:

1. **Construcción de una solución global para una familia de programas**, denominada "modelo generativo" o "línea de productos". El primer paso de esta etapa es determinar el alcance del dominio y analizar los aspectos comunes y variables entre los miembros de una familia. Para ello, se recomienda utilizar alguna de las metodologías de análisis de dominio existentes, como FODA o FAST. Cuando la complejidad de los programas de una familia es alta, se aconseja descomponer el dominio según diferentes puntos de vista (tanto conceptuales como de nivel de abstracción).

2. **Obtención de un programa particular de la familia**. La solución colectiva debe gozar de un grado de ocultación de caja negra. Esto, en el caso de la GP y las factorías de software, se consigue mediante DSLs, y en el caso de la MDA, se logra con lenguajes de modelado (UML, nuevos perfiles UML o nuevos lenguajes de modelado descritos con MOF).

La figura 2.20 relaciona la terminología, prácticamente equivalente, de la GP, las factorías de software y MDA.

[28] Análogas a los dos procesos básicos de desarrollo de un marco de trabajo descritos en la sección 2.2.3

GP [CE00]	Factorías de software [GS04]	MDA [MSUW04]
Espacio de la solución Componentes software	*Plantilla de la factoría* Componentes software	No especificado
Espacio del problema DSL	*Plantilla de la factoría* DSL	Lenguaje de modelado (especialización de UML para un dominio particular mediante un perfil o creación de un nuevo lenguaje con MOF)
Conocimiento de configuración Transformador DSL → Código final	*Plantilla de la factoría* Transformador DSL → Código final	Transformadores PIM → PSM → Código final

Figura 2.20. Equivalencia terminológica entre la GP, las factorías de software y MDA.

2.3. Compiladores de DSLs

Lamentablemente, la terminología que se usa en la transformación de programas no está estandarizada. A continuación, se citan algunas definiciones bastante difundidas:

Un ***compilador*** *es un programa que lee un programa escrito en un lenguaje, el lenguaje "fuente", y lo traduce a un programa equivalente en otro lenguaje, el lenguaje "objeto".*

[ASU90, página 1]

Un ***generador*** *es un programa que partir de una especificación abstracta de software produce su implementación.*[29, 30]

[CE00, página 333]

---

[29] *A* ***generator*** *is a program that takes a higher-level specification of a piece of software and produces its implementation.*

[30] En las páginas 341-343 del mismo texto, K. Czarnecki amplía su definición indicando que, además de los generadores de composición (*compositional generators*), que transforman en vertical (entre programas de distinto nivel de abstracción), existen los generadores de transformación (*transformational generators*), que realizan transformaciones horizontales (entre programas del mismo nivel de abstracción) y oblicuas (combinación de transformaciones verticales y horizontales).

Una **conversión** *entre modelos toma como punto de partida uno o varios modelos fuente y produce como salida otro modelo objeto.*[31]

[MSUW04, página 16]

Las técnicas de reestructuración de código pueden clasificarse como horizontales o verticales. El término **transformación** *generalmente se utiliza para identificar las transformaciones horizontales (entre artefactos del mismo nivel de abstracción). Para las transformaciones verticales, son más apropiados los términos* **traducción** *o* **síntesis***, ya que designan la creación de nuevos artefactos a partir de descripciones escritas a otro nivel de abstracción.*[32]

[Gra04]

Una **transformación** *es un proceso que, a partir de una o más especificaciones de entrada, crea o modifica una o varias especificaciones de salida. Como las especificaciones de entrada se expresan en un lenguaje del dominio de entrada, y las especificaciones de salida se expresan en un lenguaje del dominio de salida, puede considerarse que una transformación es la creación de una instancia de una relación entre los dominios de entrada y salida. Dicha relación se denomina* **conversión***.*[33]

[GS04, página 456]

Como puede apreciarse, en ocasiones se utilizan distintos nombres para el mismo concepto o para conceptos muy similares (compilación [ASU90], generación [CE00], conversión [MSUW04] y transformación [GS04] pueden considerarse sinónimos). Otras veces, se usa el mismo término con distintos significados (según [ASU90], un compilador consta fundamentalmente de dos etapas, una de análisis y otra de generación o síntesis;

---

[31] *A* **mapping** *between models is assumed to take one or more models as its input (these are the "sources") and produce one output model (this is the "target").*

[32] *Software restructuring techniques can be categorized as either horizontal or vertical. The research into horizontal transformation concerns modification of a software artifact at the same abstraction level. This is the typical connotation when one thinks of the term* **transformation***. In contrast, vertical transformation is typically more appropriately called* **translation** *or* **synthesis** *because a new artifact is being synthesized from a description at a different abstraction level.*

[33] *A* **transformation** *is a process that creates or modifies one or more output specifications, from one or more input specifications. Since each input specification is expressed in the language of an input domain, and each output specification is expressed in the language of an output domain, we can think of a transformation as creating an instance of a relationship between the input and out domains. This relationship is called a* **mapping.**

según [CE00], un generador es equivalente a un compilador; según [Gra04], la síntesis es un tipo de transformación).

Con el propósito de simplificar y evitar confusión, en esta tesis se utilizará el término tradicional de **compilación** y se adoptará la definición más amplia: *la conversión de uno o varios programas fuente en uno o varios programas objeto*. Además, se asumirá que la compilación de programas consta fundamentalmente de dos etapas: una de análisis de los programas fuente y otra de generación de los programas objeto.

La compilación puede ser manual, automática o interactiva (supervisada por el usuario del transformador) [GS04]. Esta tesis se centra exclusivamente en la compilación automática.

Como se indicó anteriormente, en las factorías de software y MDA se contempla que el resultado de una compilación sea un programa completo (corriente traduccionista) o un programa parcial (corriente elaboracionista). Cuando el producto es código ejecutable, pueden considerarse las siguientes alternativas [Cle01, Her03]:

- El código objeto ha de ser fácilmente manipulable y, por lo tanto, legible.
- Hay que evitar la manipulación directa del código objeto. Cualquier cambio se especificará en el modelo fuente, que se volverá a traducir automáticamente para actualizar el código objeto.

Fundamentalmente, existen tres **formas de construir un compilador**:

1. **Desarrollarlo desde cero como si fuera un programa aislado**. Esta opción es la más versátil, ya que permite construir cualquier compilador, pero también la más costosa. Es la alternativa escogida para el desarrollo de la práctica totalidad de los compiladores que traducen código GPL a código máquina. La figura 2.21 resume la arquitectura estándar de este tipo de compilador [ASU90]. En las fases de análisis léxico y sintáctico pueden reutilizarse, en el caso de que el programa fuente sea texto, herramientas de generación automática de analizadores como lex y yacc [BLM92] o, en el caso de que el programa fuente tenga formato XML, librerías para el manejo de árboles XML [Har02].
2. **Embeber el DSL fuente en el lenguaje de implementación del compilador**. La finalidad de un DSL es ser próximo a sus usuarios. Dado su alto nivel de abstracción, conviene que la compilación *DSL*→*Código Máquina* se apoye en el código intermedio

de algún GPL para el que haya disponible un compilador *GPL*→*Código Máquina*. De esta manera, el trabajo se reduce a construir un compilador *DSL*→*GPL* (figura 2.22)[34]. Si la sintaxis abstracta del DSL es sencilla, puede construirse su sintaxis concreta respetando la sintaxis concreta del lenguaje de implementación del compilador *DSL*→*GPL*. Es decir, para ahorrar la fase de análisis se busca embeber el DSL dentro del lenguaje de implementación del compilador, de conseguir que el usuario tenga la sensación de utilizar un DSL abstracto cuando en realidad escribe código en el mismo lenguaje con que se implementa el compilador *DSL*→*GPL*. M. Fowler [Fow05] denomina a este tipo de DSLs "internos".

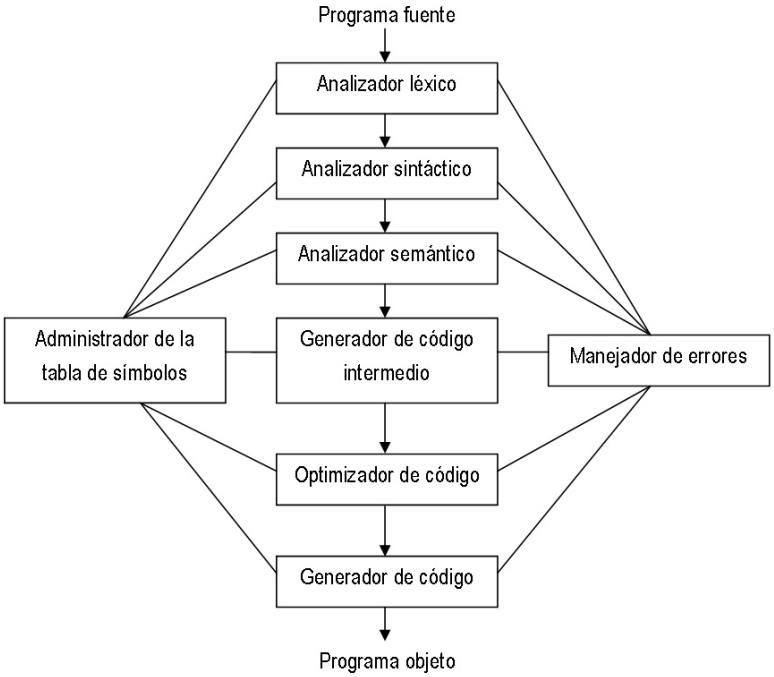


Figura 2.21. Arquitectura de un compilador según [ASU90]

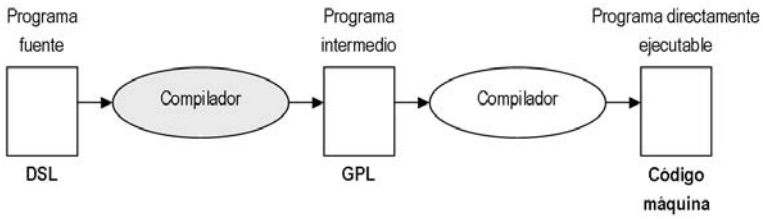


Figura 2.22. Compilador que utiliza código GPL como paso intermedio.

En [Fow05] y [Eck03, capítulo "*Multiple languages*"] se indica cómo utilizar Ruby [Rub07] y Python [Pyt07] respectivamente para construir este tipo de compiladores. Ruby es un lenguaje interpretado y con tipado dinámico que posee una sintaxis

[34] En las figuras 2.23, 2.24 y 2.25, aparece en gris el software que debería desarrollar el autor de un compilador *DSL*→*Código Máquina*.

flexible[35] y concisa, apta para la construcción de pequeños DSLs. La primera sentencia de un compilador de este tipo es leer y ejecutar el código fuente DSL, cargándose así en memoria toda la información necesaria para etapa de generación (figura 2.23). Es como si el código fuente fuera una "macro" que se invoca desde el compilador en tiempo de ejecución. El análisis del código fuente pasa a ser responsabilidad del intérprete de Ruby. Si acaso, el autor del compilador deberá describir parte del análisis semántico.

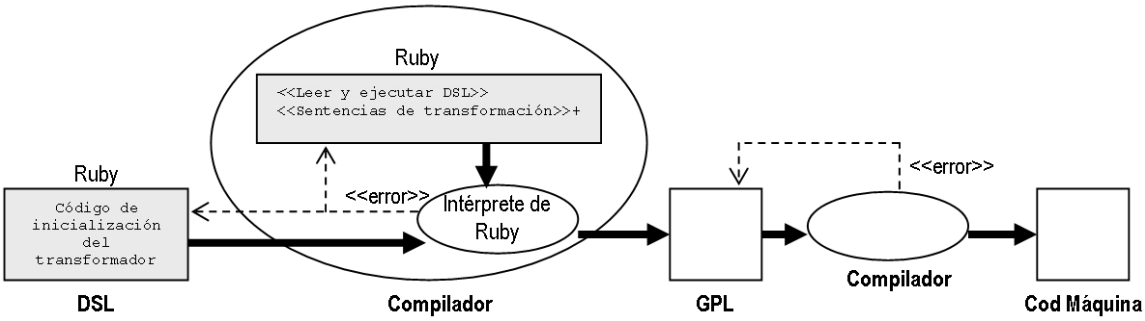


Figura 2.23. Implementación en Ruby de un compilador con DSL "interno"

Lamentablemente, como se aprecia en la figura 2.23, si se da algún error en el código fuente DSL que no es detectado por el intérprete de Ruby[36], y este error conlleva la generación de código GPL erróneo que sí es detectado por el compilador *GPL→Código Máquina*, el mensaje de error no hará referencia al código DSL, sino al código GPL intermedio, dificultándose la localización del error original. En ocasiones, si el DSL y el GPL son internos, es decir, si coinciden el DSL, el GPL y el lenguaje de implementación del compilador *DSL→GPL*, puede conseguirse que los mensajes de error referencien al código fuente DSL. En [CE00, capítulo 10] se describe como conseguir esta prestación en C++, utilizando sus plantillas para implementar compiladores que reciben DSLs con sintaxis C++ y producen código intermedio C++ (figura 2.24).

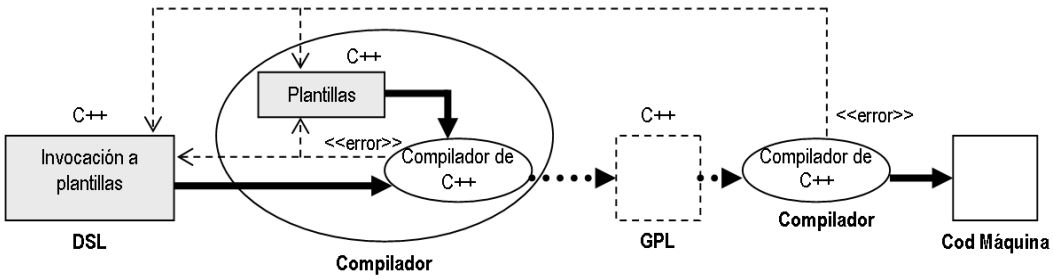


Figura 2.24. Implementación en C++ de un compilador con DSL y GPL "internos"

[35] Con abundante *syntactic sugar*.

[36] ni por la posible comprobación semántica del compilador.

Evidentemente, el punto débil de esta manera de construir un compilador es hacer depender la sintaxis concreta del DSL de la sintaxis concreta del lenguaje del compilador, limitando la expresividad del DSL.

3. **Aprovechar una infraestructura propia para la construcción de compiladores**. Dentro del marco de las factorías de software y MDA, están surgiendo herramientas que facilitan la creación de DSLs y su traducción a código GPL ejecutable. Un elemento clave de estas herramientas es el lenguaje con que se expresan las transformaciones *DSL*→*GPL*.

2.3.1. Lenguajes de transformación

Los lenguajes de transformación suelen clasificarse [CH03, MSUW04, QVT05] en:

– **Declarativos o relacionales**. Expresan reglas de transformación que establecen vínculos entre patrones del lenguaje fuente y del lenguaje objeto. Las reglas, por tanto, son correspondencias bidireccionales que posibilitan la doble transformación *fuente*→*objeto* y *objeto*→*fuente*. Según la representación de los programas y cómo se expresen los patrones, puede distinguirse:

  - La utilización de lenguajes de modelado tipo UML para representar los programas y de lenguajes de restricciones tipo OCL para expresar los patrones [AK02, CDI03, Mil02].

  - El uso de grafos para representar los programas y de subgrafos para expresar los patrones [AEHH96, BM03, VVP02].

– **Imperativos u operacionales**. Especifican cómo obtener el programa objeto a partir del programa fuente. Lamentablemente, las transformaciones son unidireccionales (*fuente*→*objeto*). Puede distinguirse:

  - Recorrer el programa fuente y, a medida que se recorre, ir generando el programa objeto. La versión más simple de este enfoque es la "traducción dirigida por la sintaxis" descrita en [ASU90] donde, según se atraviesa el árbol sintáctico del programa fuente, se escribe el programa objeto en un flujo de texto (*text stream)*. Con el fin de facilitar el recorrido del programa fuente, se puede aprovechar la tecnología XSL [PBG01, DHO01] o aplicar el patrón de diseño *Visitor* [GHJV94], como hace el marco de trabajo Jamda [Jam06].

- Utilizar plantillas[37] como soporte de la producción del programa objeto. Una plantilla puede considerarse como un "programa con huecos" que se rellenan a partir de la información contenida en el programa fuente. Esta aproximación es especialmente adecuada para la generación de texto [MSUW04] y se utiliza en:

  - ► Muchas herramientas MDA, por ejemplo, *AndroMDA* [AM06], *CodaGen Architect* [CA06]…

  - ► La generación de páginas web, como por ejemplo, ASP (*Active Server Pages*) [Wei00] o JSP (*Java Server Pages*) [Ber03]. Estas tecnologías han sido el punto de partida de otras herramientas de producción de plantillas, como Jostraca [Jos04] o ERB [ERB06], aptas para cualquier lenguaje objeto textual.

Además, existen lenguajes híbridos declarativo-imperativos como ATL (*Atlas Transformation Language*) [BDJR03], TXL [TXL06] o **QVT** (*Query/View/Transformation*) [QVT05]. Este último, es el lenguaje propuesto por OMG para la transformación entre modelos MOF en el marco de la MDA. Consta de dos partes (figura 2.25):

- **Parte declarativa**. Está formada por un lenguaje de alto nivel de abstracción para expresar relaciones (*The Relations Language*) y por un lenguaje base menos abstracto semánticamente equivalente (*The Core Language*). Teóricamente, el usuario de QVT sólo utiliza el lenguaje de relaciones, que se traduce automáticamente al lenguaje base.

- **Parte imperativa**. Para los casos en que es difícil especificar una transformación en declarativo, QVT ofrece el lenguaje operacional de conversiones (*The Operational Mappings Language*). Además, QVT permite la invocación de transformaciones externas elaboradas con otros lenguajes mediante una interfaz de caja negra (*Black Box Implementations*).

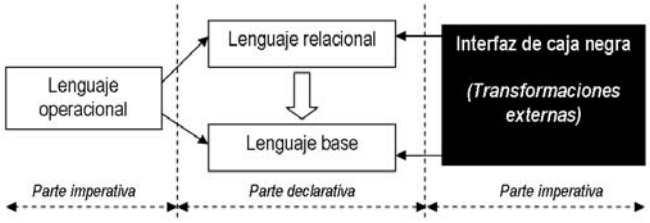


Figura 2.25. Arquitectura de QVT.

[37] Algunos autores [MSUW04, páginas 52-53], utilizan el término "arquetipo" en lugar de "plantilla".

Actualmente, están apareciendo las primeras implementaciones de QVT [SQVT0, TArch0, Xac07].

3

Proceso EDD para el desarrollo de familias de productos

When I'm doing templating I usually like to first write a hard coded class for a single case, get that class working and debugged, and then (as gradually as I can) replace the hard coded elements with templated elements.

M. Fowler, *Generating code for DSLs.*

EDD es un proceso de desarrollo que plantea la idea original de aprovechar la similitud entre los productos de una familia para construirlos por analogía. La figura 3.1 da una visión panorámica de EDD, cuya primera actividad es realizar un producto de la familia. A continuación, se busca cómo **flexibilizar** este **ejemplar** para que satisfaga los requisitos del resto de los productos. Es decir, se trata de definir una relación de analogía que permita derivar los demás productos del ejemplar. Por último, se obtienen todos los productos aprovechando la flexibilización del ejemplar.

Para que la obtención de los productos sea automática, resulta esencial que la flexibilización del ejemplar sea formal. Este objetivo se podrá alcanzar en mayor o menor medida según el grado de formalización del propio ejemplar. Por ejemplo, en el capítulo 2 se señaló que los productos del análisis y el diseño no suelen ser formales. Como resultado, la derivación automática por analogía de dichos productos será muy limitada. En cambio, los productos de la codificación y las pruebas, que suelen ser formales, sí se podrán obtener

automáticamente. Para que el lector se haga una idea de la potencia y versatilidad de EDD, los capítulos 5 y 6 de esta tesis incluyen ejemplos de derivación automática por analogía de programas escritos en Java, C++ y TRANSACT SQL; de juegos de prueba escritos en Java y Modula-2; y de documentación escrita en HTML y Javadoc.

Las siguientes secciones describen EDD en profundidad.

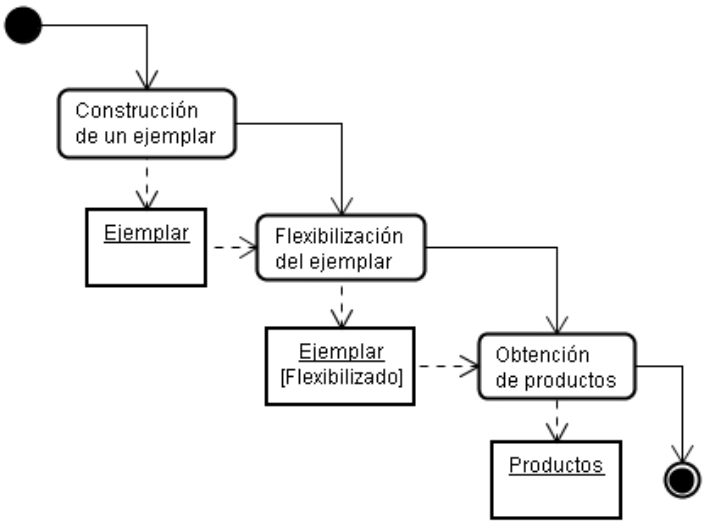


Figura 3.1. Visión panorámica de EDD (notación: diagrama de actividades UML).

3.1. Construcción de un ejemplar

Al comienzo del desarrollo de una familia de productos suele disponerse de algún ejemplar, pues la decisión de elaborar una familia a menudo se toma al detectar trabajo repetitivo en el desarrollo aislado de varios productos de un dominio o al identificar oportunidades de negocio en la ampliación de las prestaciones de un producto de éxito [MRS02]. Precisamente, una de las ventajas de EDD frente a otros procesos de desarrollo[38] es que reconoce esta situación y trata de aprovecharla mediante la reutilización íntegra de un ejemplar.

En los casos excepcionales en los que no se cuente con ningún ejemplar se desarrollará uno, aplicando el proceso de ingeniería del software "para productos aislados" que se estime más conveniente. En la sección 3.2.6 se indica cómo escoger de entre los requisitos de una familia de productos, un conjunto representativo para el ejemplar.

[38] En la sección 2.2 se revisan tres procesos de desarrollo de familias de productos que actualmente cuentan con una gran difusión.

3.2. Flexibilización del ejemplar

La flexibilización del ejemplar, como indica la figura 3.2, se descompone en 1) el análisis de los requisitos de la familia de productos, 2) la definición de una interfaz que facilite la especificación abstracta de los productos de la familia y 3) la implementación de la flexibilización, que a partir de las especificaciones abstractas obtenga los productos finales correspondientes.

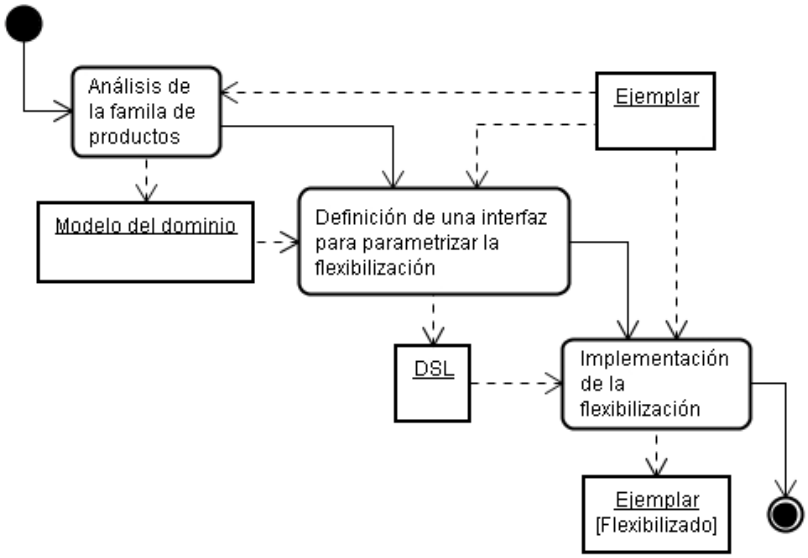


Figura 3.2. Flexibilización de un ejemplar (notación: diagrama de actividades UML).

3.2.1. Análisis de la familia de productos

Los objetivos del análisis de una familia de productos software son:

- Definir y acotar la familia.
- Recabar toda la información relevante del dominio e integrarla de forma coherente en un modelo.

El análisis de una familia de productos se descompone en 1) la definición del dominio de la familia, 2) la identificación y clasificación de los requisitos de la familia, 3) el modelado del dominio, 4) el análisis de la rentabilidad de la flexibilización del ejemplar y 5) el ajuste del modelo del domino.

3.2.1.1. Definición del dominio

Aprovechando la especificación de requisitos del ejemplar, se procederá a:

- Identificar a los clientes potenciales de la familia de productos, sus necesidades y expectativas.

- Describir, de forma aproximada, el tipo de productos que se desea obtener.
- Describir el contexto donde se emplearán los productos. Esta actividad es especialmente importante en el caso de que se quiera producir componentes de otros sistemas y puedan darse problemas de incompatibilidad.
- Recoger el vocabulario propio del dominio en un glosario de términos.

3.2.1.2. Identificación y clasificación de los requisitos de la familia de productos

Utilizando la definición del dominio, se identificarán los requisitos de los productos de la familia, anotando su nivel de importancia según la demanda de los clientes potenciales.

A continuación, los requisitos se clasificarán en fijos y variables. Un requisito es fijo si es común a todos los productos de la familia y tiene un valor único. Un requisito es variable si no es fijo. Si se dispone de más de un ejemplar, la búsqueda de los requisitos fijos comenzará examinando la intersección entre los requisitos de los ejemplares.

Interesa encontrar un equilibrio entre la cantidad de requisitos fijos y variables[39]. Los requisitos fijos son clave para mejorar la economía de alcance y la abstracción de la interfaz con que se especificarán los productos, que se limitará al ajuste de los requisitos variables. Por otro lado, el aumento de los requisitos variables amplía el dominio y, por tanto, los clientes potenciales de la familia. La decisión de considerar como fijo o variable un requisito se revisará durante la tarea 3.2.1.5, donde se ajustará el modelo del dominio.

Conviene determinar las siguientes propiedades de cada requisito variable:

- Su rango de variación.
- Si tiene dependencia con otros requisitos. Por ejemplo, si su valor se deriva del valor de otros requisitos, si ciertas combinaciones de valores están prohibidas…
- El momento en que se concretará su valor (*binding time*).
- Quién especifica el valor de cada requisito (*binding role*).

[39] Este fenómeno se expone con claridad en [Cle01, sección 2.9, "*A Balancing Act*"].

3.2.1.3. Modelado del dominio

Toda la información anterior se recopilará en un modelo del dominio. Siguiendo las recomendaciones de [CE00] y [GS04], sugerimos utilizar el modelado de características (*feature modeling*) integrado en la metodología FODA (*Feature-Oriented Domain Analysis*) [KCHN+90] y resumido en la sección 2.2.1.

3.2.1.4. Análisis de la rentabilidad de la flexibilización del ejemplar

Para decidir si vale la pena flexibilizar el ejemplar o, por el contrario, es más conveniente construir cada producto por separado, se estimarán los costes de desarrollar y mantener:

- Cada producto por separado. Los costes de cada producto pueden estimarse por analogía con los costes del ejemplar, siguiendo las indicaciones de W. Myers [Mye89].

- La flexibilización del ejemplar y los productos obtenidos por ésta.

3.2.1.5. Ajuste del modelo del domino

Teniendo en cuenta el análisis de rentabilidad anterior, se reajustará el dominio para suprimir requisitos excesivamente costosos, pasar a considerar como fijos algunos requisitos variables…

3.2.1.6. Selección de los requisitos de un ejemplar

Como se indicó en la sección 3.2.1, cuando no exista ningún ejemplar será necesario desarrollar uno. Convendrá que el nuevo ejemplar, para facilitar su posterior flexibilización, satisfaga un conjunto representativo de los requisitos de la familia de productos. Si el modelo del dominio se describe con un diagrama FODA, dicho conjunto estará formado por:

- Todas las características obligatorias.

- Todas las características opcionales. En el caso de que las características opcionales pertenezcan a un grupo "*or* exclusivo", como máximo podría seleccionarse una característica (ver siguiente párrafo).

- Una de las ramas de cada grupo "*or* exlusivo".

- Todas las ramas de cada grupo "*or* inclusivo".

3.2.2. Definición de una interfaz para la especificación abstracta de los productos

Para facilitar la última actividad de EDD, la obtención de productos parametrizando la flexibilización del ejemplar, será conveniente que dicha flexibilización ofrezca una interfaz que oculte los detalles de implementación. EDD propone modelar esta interfaz como un DSL.

3.2.2.1. Especificación de un DSL

La especificación de un DSL, como la de cualquier lenguaje formal, comprende la definición de su sintaxis y de su semántica.

3.2.2.1.1. Sintaxis

Generalmente, la sintaxis de un DSL se definirá con una gramática independiente del contexto [ASU90] o un metamodelo [MOF06]. EDD propone el siguiente proceso para la obtención sistemática de la sintaxis abstracta de un DSL, especificada con una gramática independiente del contexto en notación EBNF (*Extended Backus-Naur Form*), a partir del modelo FODA de un dominio. Con el propósito de aclarar este proceso se utilizará como ejemplo el diagrama de la figura 3.3.

Paso 1. Poda de los nodos fijos del diagrama FODA

La principal razón por la que los DSLs pueden ser más abstractos que los GPLs es porque, al restringir su área de aplicación a un dominio, se elimina la necesidad de describir los requisitos fijos cada vez que se especifique un producto. Por ello, el primer paso de la traducción *FODA*→*EBNF* será suprimir los nodos "fijos" del diagrama FODA, que se determinarán del siguiente modo:

1) Todos los nodos hoja obligatorios (con cardinalidad [1..1]) son fijos.

2) Un nodo, que no sea hoja, es fijo si es obligatorio y todos sus descendientes son fijos.

Por ejemplo, los nodos 2, 5, 11, 12, 17 y 18 de la figura 3.3 son fijos por ser hoja y obligatorios. El nodo 10 también es fijo porque es obligatorio y todos sus descendientes, los nodos 11 y 12, son fijos. Sin embargo, el nodo 1 no es fijo pues, a pesar de ser obligatorio, su descendiente 3 no es fijo.

La figura 3.4 es el diagrama FODA podado.

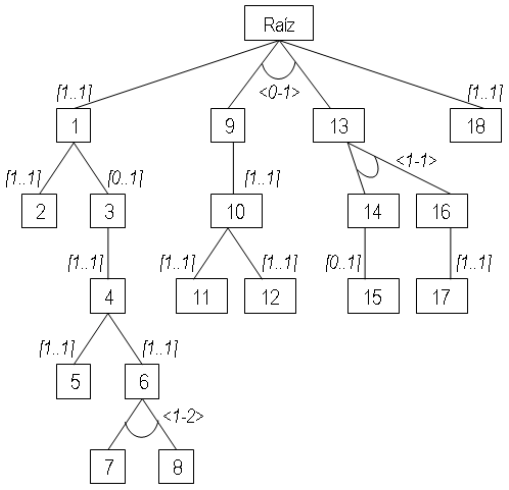


Figura 3.3. Ejemplo de modelo de un dominio.

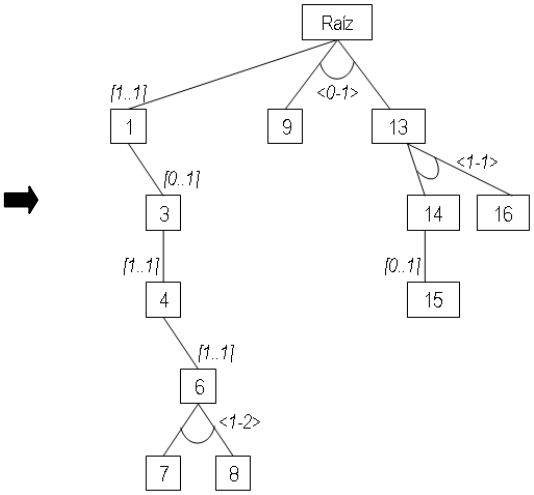


Figura 3.4. Modelo podado de un dominio.

Paso 2. Conversión de las características FODA en producciones EBNF

Aplicando la tabla de conversión de la figura 3.5, se traducirán las características FODA en producciones EBNF[40]. La raíz del árbol FODA será el símbolo inicial de la gramática y las hojas, los símbolos terminales.

La figura 3.6 es la gramática resultante de aplicar las conversiones al diagrama de la figura 3.4. La figura 3.7 es un ejemplo de especificación en el DSL descrito por la gramática. La figura 3.8 es el árbol sintáctico resultante de analizar la especificación del ejemplo. Como puede apreciarse, el grado de abstracción alcanzado es significativo. El código necesario para especificar un producto con las características 1, 2, 3, 4, 5, 6, 7, 8, 13, 14, 15 y 18, tan sólo menciona las características 7, 8 y 15, lo que supone un "ahorro" del 75%. Las características 1, 2, 3, 4, 5, 6, 13, 14 y 18 están implícitas y se deducen automáticamente.

---

[40] En esta tesis las gramáticas se escribirán según la notación EBNF de *ProGrammar* [PGram06], un entorno visual para el desarrollo de analizadores sintácticos (*parsers*). En *ProGrammar* :

- las reglas de producción se delimitan con puntos y comas.
- las llaves {} indican repetición *1:N*.
- los símbolos terminales son cadenas de texto entrecomilladas.

FODA	A [1..1] B	A [0..1] B	A [1..N] B	A [0..N] B	A <1-1> B C	A <1-2> B C	A <0-1> B C
EBNF	A ::= B	A ::= [B]	A ::= {B}	A ::= [{B}]	A ::= B | C	A ::= B | C | B C	A ::= [B] | [C]

Figura 3.5. Tabla de conversión FODA → EBNF.

```
Raiz ::= Nodo_1 (["Nodo_9"] | [Nodo_13]);
Nodo_1 ::= [Nodo_3];
Nodo_3 ::= Nodo_4;
Nodo_4 ::= Nodo_6;
Nodo_6 ::= "Nodo_7" | "Nodo_8" | "Nodo_7" "Nodo_8";
Nodo_13 ::= Nodo_14 | "Nodo_16";
Nodo_14 ::= ["Nodo_15"];
```

Figura 3.6. Gramática independiente del contexto equivalente al modelo de la figura 3.4.

```
Nodo_7 Nodo_8 Nodo_15
```

Figura 3.7. Ejemplo de especificación.

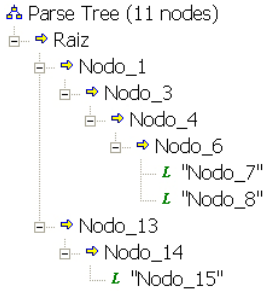


Figura 3.8. Árbol sintáctico correspondiente a la especificación de la figura 3.7.

Con el fin de facilitar la edición del código y mejorar su legibilidad, algunos autores [Fow05, GS04, IS06, Jet06] defienden la oferta de distintas representaciones de un mismo lenguaje orientadas a diferentes situaciones (por ejemplo, la posibilidad de que un programa C++ pueda editarse como un documento XML o como un árbol sintáctico). Estos autores suelen distinguir entre la **sintaxis abstracta** de un lenguaje, que caracteriza conceptualmente los elementos del lenguaje y las reglas que rigen su combinación, y sus **sintaxis concretas**, que definen diversas representaciones [GS04, páginas 280 y 281]. En esta tesis se exploran las siguientes opciones a la hora de definir sintaxis concretas:

- Utilizar un formato XML que facilite el análisis de la especificación de los productos. Esa alternativa se aplica en los ejemplos incluidos en las secciones 4.1.4.1 y 6.2.

- Hacer coincidir la sintaxis concreta del DSL con la sintaxis del lenguaje con que se implementa la flexibilización del ejemplar. Así, se evita el análisis léxico y

sintáctico de la especificación de los productos. Esta alternativa de "DSLs internos" se aplica en los ejemplos resueltos en las secciones 5.1.2.3 y 5.2.2.

- Hacer coincidir la sintaxis concreta del DSL con la sintaxis del lenguaje de implementación del ejemplar. Esta posibilidad de "DSLs embebidos" facilita la extensión del lenguaje de implementación del ejemplar. En las secciones 6.1, 6.4 y 6.5 se motiva y ejemplifica este tipo de DSLs.

- Sustituir el DSL por metainformación. En ocasiones, la parametrización de la flexibilización del ejemplar puede deducirse automáticamente consultando metainformación. Esta alternativa supone el máximo nivel de automatización y se ejemplifica en la sección 6.3.

3.2.2.1.2. Semántica

Para definir la semántica de un DSL, J. Greenfield y K. Short [GS04, páginas 309-318] recomiendan dos alternativas:

a) Expresar una traducción del DSL a otro lenguaje cuya semántica esté bien definida.

b) Expresar la semántica del DSL en función del hilo de ejecución de las sentencias del DSL.

Cuando los productos de una familia y la flexibilización del ejemplar sean formales, la semántica del DSL vendrá implícita en dicha flexibilización, que describirá como obtener un producto final a partir de su especificación DSL (opción a).

3.2.2.2. Cualidades deseables de un DSL

Conviene que un DSL, como todo lenguaje, satisfaga las siguientes propiedades:

- **Integridad conceptual** [GS04, PZ98]. Un lenguaje debe proporcionar un marco conceptual para pensar acerca de los problemas que se tratan de resolver y un medio para expresar su solución. El DSL incorporará, en la medida de lo posible, la terminología y los conceptos básicos del dominio recogidos en la "definición del dominio".

- **Manejabilidad** [GS04]. La gramática concreta del DSL será fácil de utilizar.

- **Debe promover la escritura de un código conciso y legible** [Lou04].

- **Regularidad** [Lou04]. Un lenguaje debe contemplar cualquier combinación conceptualmente coherente de sus primitivas. La regularidad facilita el aprendizaje del lenguaje y la escritura de código porque reduce las excepciones o casos especiales que deben recordarse

- **Fiabilidad** [AV98]. Un lenguaje debe ofrecer mecanismos para detectar un mal uso del mismo. Un ejemplo de mecanismo de control de errores, implantado en la mayoría de los GPLs, es el sistema de tipos.

- **Consistencia con notaciones aceptadas convencionalmente** [Lou04]**.** Por ejemplo, algo que conceptualmente es una suma debería denotarse con el símbolo +.

3.2.3. Implementación de la flexibilización del ejemplar

Para la flexibilización de un ejemplar, deberá disponerse de algún mecanismo formal que cumpla las propiedades enunciadas en la sección 1.3.2, es decir, que sea modular, no invasivo, aplicable a la flexibilización de cualquier producto software, capaz de producir productos eficientes y con algún medio para detectar automáticamente errores en la flexibilización.

En la sección 2.1.3 se señaló que el código es el producto del ciclo de vida con mayor tasa de reutilización. Por esta razón, las técnicas que comúnmente se emplean para generalizar código y posibilitar su reutilización parecen buenas candidatas para la flexibilización de un ejemplar. Concretamente, en esta tesis se examinará la viabilidad de las técnicas recogidas en la figura 3.9. En nuestro estudio, distinguiremos entre técnicas internas y externas.

- Las **técnicas internas** son aquellas que están disponibles en el lenguaje de codificación del ejemplar. Por ejemplo, si el ejemplar es un programa escrito en Java, las técnicas internas son la herencia de clases, el polimorfismo… Al utilizar técnicas internas se aprovechan los medios del lenguaje para la detección automática de errores. Por ejemplo, un "error de tipos" en la flexibilización anterior sería detectado y localizado correctamente por el "compilador" de Java (*javac*). Lamentablemente, las técnicas internas padecen los siguientes inconvenientes:

  a) Normalmente, sólo están disponibles en los GPLs. Por ejemplo, la flexibilización de código HTML no podría realizarse con herencia.

		Ámbito de la flexibilización	Modularidad	No invasividad	Ajuste de la variabilidad en tiempo de compilación
Técnicas internas	**Variables**	Literales	X	X	X
	Sentencias de selección	Comportamiento	X	X	X
	Herencia de clases (enlace dinámico)	Tipos	✓	X	X
	Genericidad	Tipos	✓	X	✓
	Subprogramas + sentencias de selección	Comportamiento	✓	X	X
	Composición de clases + herencia de clases	Estructura y comportamiento de una clase	✓	X	X
	Composición de clases + genericidad	Estructura y comportamiento de una clase	✓	X	✓
	Sobrecarga de subprogramas	Cabecera y comportamiento de un subprograma	✓	✓	✓
	Herencia con sobreescritura	Estructura y comportamiento de una clase	✓	✓	X
	Orientación a aspectos (tejido dinámico de aspectos)	Estructura y comportamiento de una clase	✓	✓	X
	Orientación a aspectos (tejido estático de aspectos)	Estructura y comportamiento de una clase	✓	✓	✓
Técnicas externas	**Plantillas con carga de procesamiento de datos**	Cualquier fragmento de texto	X	X	✓
	Plantillas sin carga de procesamiento de datos	Cualquier fragmento de texto	✓	X	✓
	ETL	Cualquier fragmento de texto	✓	✓	✓

Figura 3.9. Técnicas de flexibilización que se evaluarán en el capítulo 5.

b) En muchas ocasiones, las técnicas internas sólo son capaces de ciertos tipos de flexibilización. Además, entre las técnicas incluidas en un GPL suelen darse solapamientos. Como consecuencia, muchas flexibilizaciones se complican porque requieren el uso combinado de distintas técnicas o la elección entre

técnicas alternativas, sin contar, en muchas ocasiones, con unos criterios claros para escoger la técnica más adecuada[41].

c) Cuando los productos finales son programas, conviene distinguir entre la variabilidad interna de cada programa y la variabilidad entre los programas. Normalmente, la variabilidad interna de un programa se ajustará o modificará (*binding time*) durante la ejecución del programa. Sin embargo, los rasgos que diferencian a un programa del resto, suelen poder ajustarse durante la creación del programa. Algunas técnicas internas no son capaces de parametrizar la variabilidad en tiempo de compilación. La gestión de la variabilidad entre programas con estas técnicas ralentizará innecesariamente los programas finales.

- Las **técnicas externas** flexibilizan un ejemplar por generación de código. Por ejemplo, la generalización de un programa escrito en Java puede conseguirse utilizando el GPL Ruby y su librería de plantillas ERB [ERB06]. Las ventajas e inconvenientes de las técnicas externas son las contrarias a las ventajas e inconvenientes de las técnicas internas. Las técnicas externas son aplicables a cualquier producto software, manejan la variabilidad entre programas en tiempo de generación y son capaces de cualquier flexibilización. Sin embargo, los medios de que disponen para el control de errores de flexibilización son muy limitados.

  XSLT (*Extensible Stylesheet Language Transformations*) [DuC01] es el lenguaje estándar propuesto por el W3C (*World Wide Web Consortium*) para transformar documentos XML en cualquier otro tipo de documentos. Aunque XML podría ser una valiosa técnica de flexibilización externa, padece serias limitaciones que nos han llevado a desestimarla:

  a) XSLT sólo puede flexibilizar ejemplares que sean documentos XML[42].

  b) La flexibilización de un ejemplar ha de poder parametrizarse con las especificaciones DSL de cada producto. Lamentablemente, XSLT no dispone de una manera estándar de pasar dicha especificación como argumento de una hoja de estilos.

  c) La modularidad y capacidad de reutilización de las hojas de estilo son limitadas (XSLT no incluye, por ejemplo, la relación de herencia entre hojas de estilo).

---

[41] La dificultad de escoger la técnica de generalización más conveniente para un problema dado se ilustra en el capítulo 6 de [Cop99], donde se trata de sistematizar esta elección.

[42] QVT también padece esta limitación, pues sólo es capaz de manipular documentos XMI.

d) La capacidad de procesamiento de datos de XSLT es muy limitada: carece de muchos de los tipos y operadores básicos de un GPL, de la capacidad de extraer información de bases de datos relacionales o de ficheros que no sean XML…

e) La sintaxis XML de XSLT es muy prolija.

La explicación sobre cómo aplicar las técnicas de la figura 3.9 en la flexibilización de un ejemplar se deja para el capítulo 5. Como se demostrará, las técnicas habituales de generalización de código padecen serias limitaciones para la flexibilización. Con el fin de superar dichas limitaciones esta tesis propone el lenguaje ETL, que se expone en el próximo capítulo.

3.2.4. Flexibilización iterativa

EDD es un proceso iterativo que afronta la dificultad de desarrollar y mantener una familia de productos mediante la flexibilización progresiva del ejemplar. Para que esto sea posible, el lenguaje ETL soporta la implementación de módulos de flexibilización que no invaden el ejemplar y que pueden superponerse a éste de forma sucesiva. La figura 3.10 integra el proceso EDD con el **modelo de ciclo de vida en espiral** propuesto por B. Boehm [Boe88]. En cada fase de "planificación" se tratará de ampliar el dominio de la familia de productos; en la fase de "análisis de riesgo" se evaluará la rentabilidad de la ampliación, examinando diferentes alternativas, seleccionando la más ventajosa y tomando precauciones para evitar futuros inconvenientes; en la fase de "ingeniería" se ampliará el modelo del dominio, la interfaz y la implementación de la flexibilización; por último, en la fase de "evaluación" se analizarán los resultados de la fase de ingeniería, examinado el grado de satisfacción obtenido en los clientes de la familia. El resultado de esta evaluación se utilizará como información de entrada para la planificación del ciclo siguiente.

Figura 3.10. Integración de EDD y ETL en el ciclo de vida en espiral.

Por otro lado, así como el desarrollo por refinamientos sucesivos [Wir71] utiliza la idea de "descomposición" para construir operaciones complejas dividiéndolas progresivamente hasta alcanzar operaciones elementales, EDD plantea la siguiente estrategia de "**analogías sucesivas**": por definición, cualquier ejemplar satisface los requisitos fijos de una familia de productos, sin embargo, con un requisito variable pueden darse las siguientes situaciones:

1. El requisito tiene un valor único y está implementado en el ejemplar.

2. El requisito toma distintos valores en el dominio y el ejemplar implementa uno de ellos. En este caso, será necesario flexibilizar la implementación del requisito para cubrir todo el rango de valores.

3. El requisito no está implementado por el ejemplar. Cuando un requisito no esté en absoluto implementado por el ejemplar, se le considerará como una "familia de productos en miniatura" sobre la que volver a aplicar EDD, desarrollando un subejemplar que satisfaga exclusivamente un valor concreto del requisito y flexibilizándolo después para que cubra el resto de los valores del requisito.

3.3. Obtención de productos a partir de la flexibilización del ejemplar

Aprovechando la interfaz abstracta de la flexibilización del ejemplar, se especificarán las particularidades de cada producto.

Aunque en los ejemplos incluidos en esta tesis el resultado de parametrizar una flexibilización es la generación automática de un producto completo, también puede optarse por la generación de productos parciales, que después se completen manualmente.

4

Lenguaje de transformaciones ETL

Language shapes the way we think
and determines what we can think about.

B.L.Whorf

El lenguaje ETL es el medio que propone esta tesis para flexibilizar ejemplares. Se trata de una técnica externa, modular, no invasiva y aplicable a cualquier producto software.

Generalmente, el resultado de flexibilizar un ejemplar con ETL será un compilador como el de la figura 4.1, compuesto por un analizador y un generador. El analizador creará una representación interna del código fuente DSL y se la pasará al generador, que obtendrá el producto final correspondiente. ETL facilitará la escritura del generador, que en realidad será un nuevo compilador (*subcompilador*) que transformará el ejemplar en el producto especificado en el código DSL. El subcompilador se compondrá de un analizador[43] (*subanalizador*) que creará una representación interna del ejemplar y de uno o varios generadores (*subgenerador 1, subgenerador 2,…, subgenerador N*) que actuarán de forma coordinada para modificar el ejemplar.

[43] Como se verá en la sección 4.2.2, la escritura de un analizador para el ejemplar será normalmente innecesaria, ya que la implementación Ruby de ETL aprovecha las expresiones regulares.

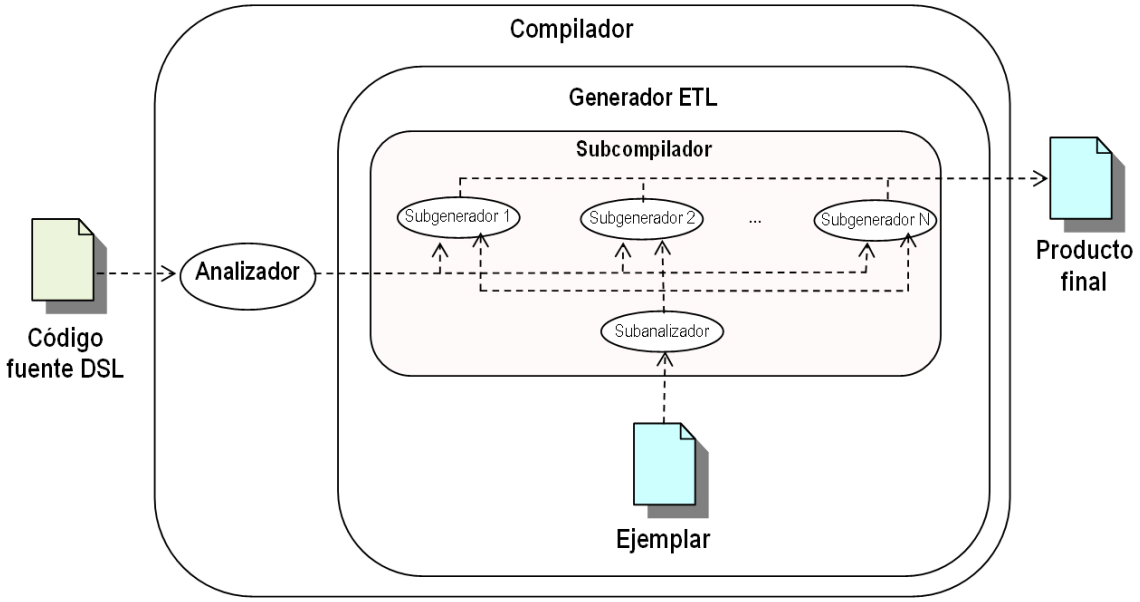


Figura 4.1. Ejemplo de compilador resultado de flexibilizar un ejemplar con ETL.

4.1. Especificación de ETL

4.1.1. Primitivas

Para desarrollar un generador ETL, se dispone de las siguientes primitivas:

1. **Sustitución**. Expresa el intercambio de un patrón de código del ejemplar por nuevo código.

   Cuando la implementación del ejemplar sea modular y esté repartida entre varios ficheros, con frecuencia será necesario expresar una misma sustitución sobre más de un fichero. Para facilitar la reutilización de las sustituciones y evitar así su repetición, las sustituciones son independientes de los ficheros del ejemplar y de los ficheros del producto final. Una sustitución se liga a los ficheros del ejemplar y del producto final mediante la primitiva de "producción", que se explica a continuación.

2. **Producción**. Expresa la aplicación simultánea[44] de un conjunto de sustituciones sobre un fichero del ejemplar para obtener un fichero del producto final. En el caso de que el conjunto sea vacío, la producción se limitará a copiar el fichero del ejemplar en el fichero del producto final.

[44] El orden en que se expresan las sustituciones de una producción es irrelevante.

Una **sustitución** puede ser **local** o **global**. Cuando sea local, se aplicará exclusivamente sobre la primera ocurrencia de su patrón de código en el fichero del ejemplar. Cuando sea global, se aplicará sobre todas las ocurrencias.

Puede suceder que varias de las sustituciones de una producción se solapen si indican cambios distintos sobre el mismo fragmento de un fichero del ejemplar. Para prevenir errores en el programa objeto, las **colisiones** entre sustituciones se detectarán automáticamente.

3. **Generación**. Expresa la ejecución simultánea[45] de un conjunto de producciones. Esta primitiva facilita la detección automática de colisiones entre las sustituciones de distintas producciones.

4.1.2. Operadores para la combinación de generadores

Dos generadores, G1 y G2, pueden combinarse mediante los siguientes operadores:

a. **Secuencia (G1, G2)**. Ejecuta primero G1 y después G2.

b. **Suma (G1, G2)**. Obtiene un nuevo generador cuyas sustituciones y producciones son la unión de las sustituciones y producciones de G1 y G2.

c. **Superposición (G1, G2)**. Actualiza las sustituciones y las producciones de G1 con las de G2. Las que tienen el mismo nombre se "sobrescriben", y las que no, se añaden.

Los operadores de suma y superposición verificarán automáticamente la ausencia de colisiones.

Los operadores pueden combinarse para conseguir generaciones complejas. Por ejemplo, `Secuencia(Suma(Superposición(G1, G2), G3), G4)`.

4.1.3. Metamodelo simplificado

La generación de código es una tarea lo suficientemente compleja como para que ETL deba incorporar los elementos básicos de un GPL orientado a objetos (variables, tipos, sentencias de selección e iteración, clases, herencia…). Sin embargo, para facilitar la comprensión de ETL, el metamodelo de la figura 4.2 sólo describe sus elementos originales.

---

[45] El orden en que se expresan las producciones de una generación es irrelevante.

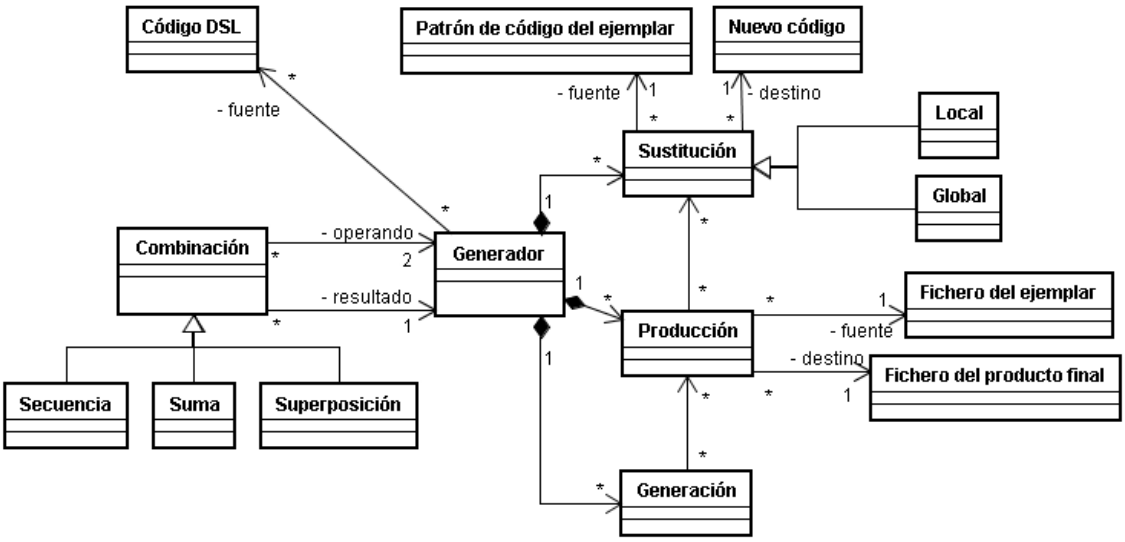


Figura 4.2. Metamodelo simplificado de ETL.

4.1.4. Ejemplo

Para ilustrar lo explicado, se utilizará un ejemplo planteado por B. Eckel en [Eck03]. Se trata de un programa escrito en Java que emula el reciclaje de basura. El programa recibe un cubo de basura con residuos de papel y aluminio mezclados, y un par de cubos donde deberá separarlos. Cada residuo tiene un peso y un valor de venta según su tipo (el papel se revende a 0.10 y el aluminio a 1,67). La figura 4.3 es el diseño del citado programa y las figuras 4.4, 4.5, 4.6 y 4.7 son la codificación. La figura 4.8 es la clase *RecycleTest*, que aprovecha el marco de trabajo *JUnit* [Jun06] para probar el programa de reciclaje. La figura 4.9 muestra un ejemplo de ejecución de *RecycleTest*.

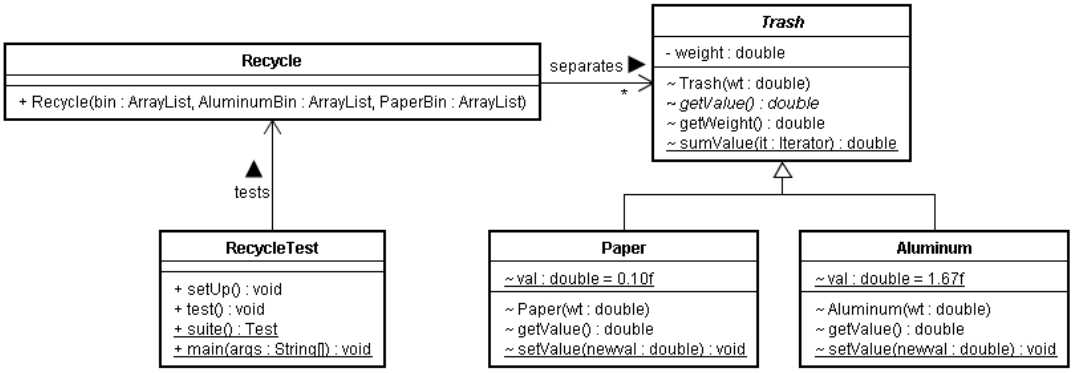


Figura 4.3. Diseño del programa de reciclaje.

```
import java.util.*;
import java.io.*;

public abstract class Trash {
  private double weight;

  Trash(double wt) { weight = wt; }

  abstract double getValue();

  double getWeight() { return weight; }

  static double sumValue(Iterator it) {
    double val = 0.0f;
    while(it.hasNext()) {
      Trash t = (Trash)it.next();
      val += t.getWeight() * t.getValue();
      System.out.println(
        "weight of " +
        t.getClass().getName() +
        " = " + t.getWeight());
    }
    System.out.println("Total value = " + val);
    return val;
  }
}
```

Figura 4.4. Trash.java.

```
class Paper extends Trash {
  static double val  = 0.10f;

  Paper(double wt) { super(wt); }

  double getValue() { return val; }

  static void setValue(double newval) {
    val = newval;
  }
}
```

Figura 4.5. Paper.java.

```
class Aluminum extends Trash {
  static double val  = 1.67f;

  Aluminum(double wt) { super(wt); }

  double getValue() { return val; }

  static void setValue(double newval) {
    val = newval;
  }
}
```

Figura 4.6. Aluminum.java.

```
import java.util.*;
import java.io.*;

class Recycle {
   public Recycle(ArrayList bin, ArrayList AluminumBin, ArrayList PaperBin){
      Iterator sorter = bin.iterator();
      while(sorter.hasNext()) {
         Object t = sorter.next();
         if(t instanceof Aluminum) AluminumBin.add(t);
         if(t instanceof Paper) PaperBin.add(t);
      }
   }
}
```

Figura 4.7. Recycle.java.

```
import junit.framework.*;
import java.util.*;

public class RecycleTest extends TestCase {

  private ArrayList bin, aluminumBin, paperBin;

    public void setUp() {
      bin = new ArrayList();
      aluminumBin = new ArrayList();
      paperBin = new ArrayList();
    } // setup

    public void test() {
      for(int i = 0; i < 30; i++)
          switch((int)(Math.random() * 2)) {
             case 0 :
                bin.add(new Aluminum(Math.random() * 100));
                break;
             case 1 :
                bin.add(new Paper(Math.random() * 100));
                break;
          } // switch
        new Recycle(bin, aluminumBin, paperBin);
        Assert.assertEquals(Trash.sumValue(bin.iterator()),
                  Trash.sumValue(aluminumBin.iterator()) +
                  Trash.sumValue(paperBin.iterator()),
                  0.001);
   } // test

  public static Test suite() {
    return new TestSuite(RecycleTest.class);
  }

  public static void main(String[] args) {
      junit.textui.TestRunner.run(suite());
  }
}
```

Figura 4.8. RecycleTest.java.

```
weight of Paper = 39.32843591620072
weight of Paper = 88.92978668194755
weight of Aluminum = 80.85452816636179
weight of Paper = 32.576243746729396
weight of Aluminum = 29.722260549957568
weight of Paper = 98.89483012221427
weight of Aluminum = 55.6311576550685
```

```
weight of Aluminum = 16.679971313159736
weight of Aluminum = 63.70369735367197
weight of Aluminum = 99.31196639953228
weight of Paper = 88.27970025954815
weight of Aluminum = 20.748603636018892
weight of Aluminum = 40.56515182960653
weight of Paper = 22.643263247923063
weight of Aluminum = 56.89150096879237
weight of Paper = 52.90995990666488
weight of Paper = 46.00152320626161
weight of Aluminum = 89.27841650068746
weight of Paper = 13.563064980212026
weight of Aluminum = 32.72989422251988
weight of Aluminum = 50.02212560938101
weight of Aluminum = 65.17721213633205
weight of Paper = 3.9418745477298067
weight of Aluminum = 53.34999494871964
weight of Aluminum = 27.134353946046595
weight of Aluminum = 46.568869492614226
weight of Paper = 70.17680422197154
weight of Paper = 59.93998861234137
weight of Aluminum = 27.31469131189175
weight of Paper = 56.47389719706791
Total value = 1496.3588429339263

weight of Aluminum = 80.85452816636179
weight of Aluminum = 29.722260549957568
weight of Aluminum = 55.6311576550685
weight of Aluminum = 16.679971313159736
weight of Aluminum = 63.70369735367197
weight of Aluminum = 99.31196639953228
weight of Aluminum = 20.748603636018892
weight of Aluminum = 40.56515182960653
weight of Aluminum = 56.89150096879237
weight of Aluminum = 89.27841650068746
weight of Aluminum = 32.72989422251988
weight of Aluminum = 50.02212560938101
weight of Aluminum = 65.17721213633205
weight of Aluminum = 53.34999494871964
weight of Aluminum = 27.134353946046595
weight of Aluminum = 46.568869492614226
weight of Aluminum = 27.31469131189175
Total value = 1428.9929046654145

weight of Paper = 39.32843591620072
weight of Paper = 88.92978668194755
weight of Paper = 32.576243746729396
weight of Paper = 98.89483012221427
weight of Paper = 88.27970025954815
weight of Paper = 22.643263247923063
weight of Paper = 52.90995990666488
weight of Paper = 46.00152320626161
weight of Paper = 13.563064980212026
weight of Paper = 3.9418745477298067
weight of Paper = 70.17680422197154
weight of Paper = 59.93998861234137
weight of Paper = 56.47389719706791
Total value = 67.36593826851193

Time: 0,701

OK (1 test)
```

Figura 4.9. Resultado de la ejecución de RecycleTest.java.

A continuación, se aplicará ETL para obtener la familia de productos "reciclaje de cualquier tipo de basura", utilizando como ejemplar el programa anterior[46].

4.1.4.1. Definición de la interfaz para parametrizar la flexibilización

Para especificar de forma abstracta los programas del dominio, se utilizará un **DSL** con la gramática de la figura 4.10. La especificación de un programa describirá el directorio donde colocar el programa objeto (`OutDir`) y los tipos de basura que deben separarse (`Trash`), indicando su nombre (`Name`) y su valor de venta (`Value`).

Para simplificar el análisis de las especificaciones DSL, se utilizará la tecnología XML. Por ejemplo, la figura 4.11 es la especificación XML de un programa que recicla los tipos de basura vidrio, plástico y cartón.

```
Specification ::= OutDir {Trash};
OutDir ::= '[a-zA-Z]+';
Trash ::= Name Value;
Name ::= '[a-zA-Z]+';
Value ::= '[0-9]+(\.[0-9]+)';
```

Figura 4.10. Gramática abstracta de un DSL para especificar programas de reciclaje de basura.

```
<specification>
  <outDir value="generation"/>
  <trash name="Glass" value="1.08"/>
  <trash name="Plastic" value="0.24"/>
  <trash name="Cardboard" value="0.8"/>
</specification>
```

Figura 4.11. Ejemplo de especificación XML de un programa de reciclaje de basura.

4.1.4.2. Implementación de la flexibilización

Para flexibilizar el ejemplar, se utilizarán dos generadores, llamados *RecycleGen* y *TestGen*, que gestionarán respectivamente la variabilidad del programa de reciclaje propiamente dicho y de su juego de pruebas. La figura 4.12 representa la actuación de estos generadores para obtener el programa objeto especificado en la figura 4.11.

a) **RecycleGen** consta de las siguientes producciones:

- **Producción 1**. Expresa cómo obtener una clase *Trash* objeto a partir de la clase *Trash* del ejemplar. Como las clases *Trash* de todos los programas objeto coinciden, la producción carece de sustituciones. Sus parámetros son:

    i. Fichero del ejemplar: *Trash.java*

    ii. Fichero del programa objeto: *Trash.java*

[46] En [Eck03] se flexibiliza el programa explotando los recursos de la orientación a objetos. En [HEAC05] se justifica porqué la flexibilización ETL es más ventajosa que la propuesta por B. Eckel.

iii. Conjunto de sustituciones: {}

- **Producciones 2**. Expresan cómo obtener las clases de basura objeto a partir de de la clase *Paper* del ejemplar. Como una producción indica una relación 1:1 entre un fichero del ejemplar y un fichero del programa objeto, serán necesarias tantas producciones como ficheros objeto se deseen generar (en el ejemplo de la figura 4.11, tres producciones expresarán la generación de los ficheros *Glass.java, Plastic.java* y *Cardboard.java*). En la sección 4.2.3.2 se verá cómo expresar este tipo de producciones de forma genérica en la implementación de ETL en Ruby mediante un iterador. Cada producción tendrá dos sustituciones indicando cómo adaptar el código variable de la figura 4.5, resaltado en los colores □ y □. Los parámetros de cada producción son:

  i. Fichero del ejemplar: *Paper.java*

  ii. Fichero del programa objeto: *Tipo de Basura.java*

  iii. Conjunto de sustituciones: {O, O}

- **Producción 3**. Expresa como obtener una clase *Recycle* objeto a partir de la clase *Recycle* del ejemplar. Dispone de dos sustituciones que indican cómo adaptar el código variable de la figura 4.7, resaltado en los colores □ y □. Sus parámetros son:

  i. Fichero del ejemplar: *Recycle.java*

  ii. Fichero del programa objeto: *Recycle.java*

  iii. Conjunto de sustituciones: {O, O}

b) **TestGen** consta de la **producción 4**, que expresa como obtener el juego de pruebas de un programa objeto a partir de la clase RecycleTest del ejemplar. Dispone de cuatro sustituciones que indican cómo adaptar el código variable de la figura 4.8, resaltado en los colores □, □, □ y □. Sus parámetros son:

i. Fichero del ejemplar: *RecycleTest.java*

ii. Fichero del programa objeto: *RecycleTest.java*

iii. Conjunto de sustituciones: {O, O, O, O}.

Para combinar los generadores *RecycleGen* y *TestGen* se utilizará el operador suma, obteniéndose como resultado un nuevo generador con las producciones 1, 2, 3 y 4.

Mediante una primitiva de generación, se expresará la ejecución de todas las producciones del nuevo generador.

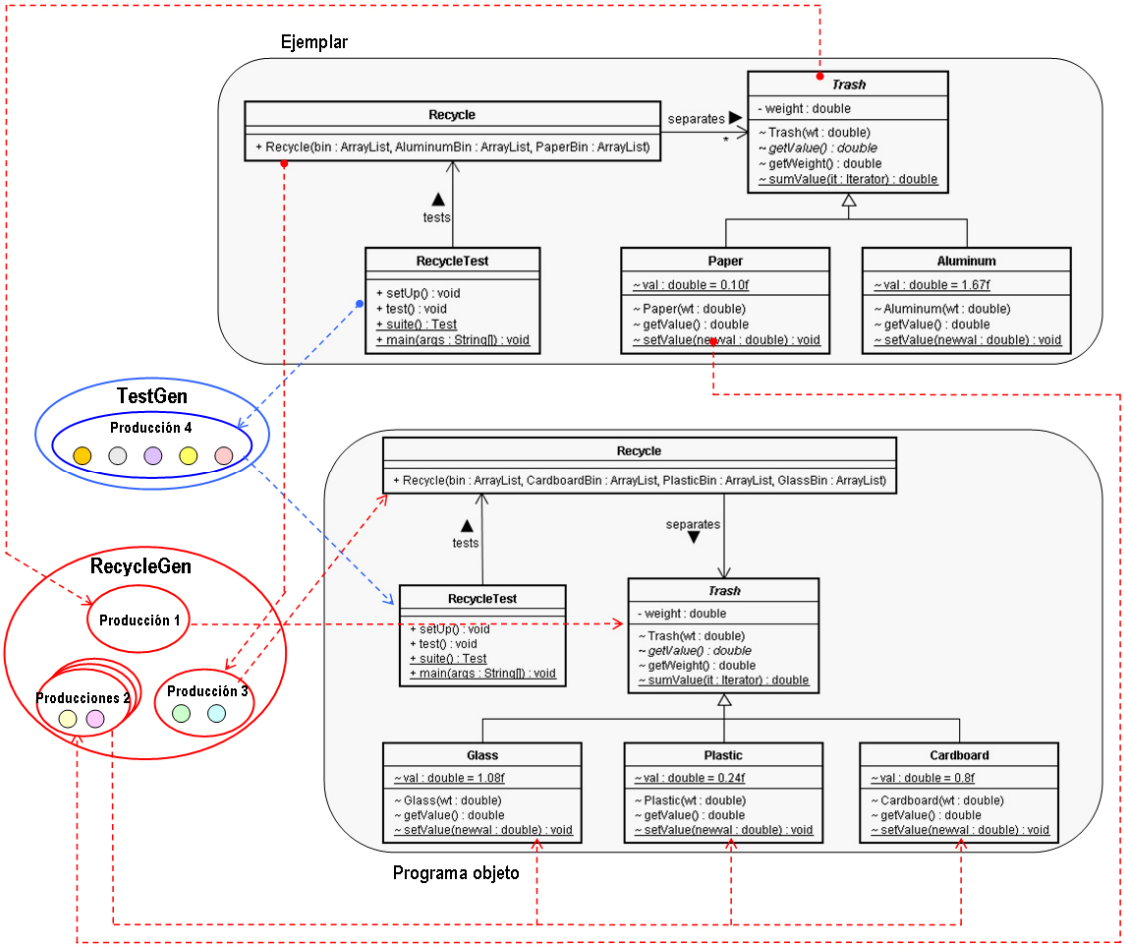


Figura 4.12. Ejemplo de generación del programa objeto especificado en la figura 4.11.

4.2. Implementación de ETL

4.2.1. Justificación de la implementación de ETL en Ruby

En un principio, se planteó implementar ETL de forma convencional, construyendo un compilador que tradujera los programas ETL en código GPL. Para agilizar el desarrollo del compilador se consideró aplicar una estrategia de arranque[47], que usara como ejemplar un programa objeto, resultado de una hipotética traducción ETL→GPL.

[47] La estrategia de arranque (*bootstrappring*) busca la escritura de un compilador $L_1 \rightarrow L_2$ en el propio lenguaje L_1. Para ello, se escribe en un lenguaje L_3 (que puede coincidir con L_2) un compilador que traduzca el núcleo de L_1, al que llamaremos L_1', en L_2. A continuación, se escribe con L_1' un compilador $L_1'' \rightarrow L_2$, donde L_1''

Lamentablemente, tras el desarrollo de algunos prototipos, se comprobó que el parecido entre los programas objeto era muy limitado. Es decir, el dominio de los "programas que generan programas" era demasiado amplio como para escribir un compilador de ETL en ETL mediante una estrategia de arranque.

Finalmente, en lugar de congelar la especificación de ETL y concentrar todos los esfuerzos en el desarrollo de un compilador tradicional, se consideró más oportuno buscar un atajo que facilitara la validación y el perfeccionamiento de ETL mediante la resolución de la mayor cantidad y variedad posible de ejemplos prácticos. Por ello, se decidió implementar ETL como un lenguaje "interno" a un GPL. Concretamente, se escogió **Ruby**[48] por las siguientes razones:

- Es muy extensible y posee una sintaxis sorprendentemente dúctil. Gracias a estas cualidades la "sintaxis" del ETL interno alcanza un grado de usabilidad razonable.
- Es totalmente Orientado a Objetos.
- Su sintaxis es muy concisa.
- Tiene una gran capacidad para la manipulación de ficheros. Concretamente, para facilitar el análisis:
    - Incorpora las expresiones regulares como tipo predefinido.
    - Dispone de analizadores XML (REXML [Rus06], XMLParser [Nas06], NQXML [Men01]…).
    - Dispone de herramientas de generación automática de analizadores (Racc [Racc05], Rockit [Rockit01]…).
    - Como Ruby es interpretado, permite el análisis directo de DSLs internos.
- Ofrece potentes contenedores (listas y tablas) e iteradores.
- Existen implementaciones del intérprete de Ruby para distintos sistemas operativos, lo que asegura un alto grado de portabilidad.

---

enriquece L_1' con nuevas instrucciones. Este proceso se repite hasta que $L_1^n=L_1$. [ASU90, páginas 743-747; Cle01, páginas 340-347]

[48] Para el aprendizaje de Ruby se recomienda el libro [TH01].

4.2.2. Implementación de las primitivas

Las primitivas de ETL están implementadas como métodos de una clase llamada *Generator*. Los generadores ETL serán clases Ruby que, al heredar de *Generator*, dispondrán de los siguientes métodos:

1. **Sustitución**. Gracias al parecido entre los productos de una familia, el análisis de un ejemplar se reducirá a la localización de pequeños fragmentos de código variable. Para evitar la escritura de analizadores para los ejemplares, se aprovechará que Ruby incluye las expresiones regulares como tipo predefinido y el patrón de código de una sustitución se expresará mediante una expresión regular[49].

   Generator posee dos métodos para definir sustituciones:

   - **`sub(regExp, text, name)`**
   - **`gsub(regExp, text, name)`**

   El parámetro `regExp` es una expresión regular que selecciona el fragmento de texto que se desea sustituir; `text` es el nuevo texto; `name` es un parámetro opcional que sirve para nombrar la sustitución. Mientras que `gsub` define una sustitución global, que intercambia todos los fragmentos de `text` que encajan con `regExp`, `sub` define una sustitución local, que sólo actúa sobre la primera ocurrencia.

2. **Producción**. Para definir producciones, *Generator* ofrece el método:

   - **`prod(iFile, oFile, subList, name)`**

   El parámetro `iFile` es un fichero del ejemplar; `oFile` es el fichero que se generará tras la aplicación de la lista `subList` de sustituciones; `subList` es opcional y si no se explicita, sobre `iFile` se aplicarán todas las sustituciones previamente definidas en el generador[50]; `name` también es opcional y sirve para nombrar la producción.

   Antes de definir una producción, `prod` verificará la ausencia de colisiones entre sus sustituciones. Si se da alguna colisión, se lanzará la excepción correspondiente.

---

[49] Para el aprendizaje de las expresiones regulares se recomienda el libro [Fri02].

[50] No dar valor a `subList` es equivalente a escribir `subList = 'all'`. Esta segunda modalidad se utilizará cuando se desee incluir en una producción todas las sustituciones previamente definidas en el generador y, además, se quiera dar valor a `name`.

3. **Generación**. Para definir una generación, *Generator* posee el método:

   - **`gen(prodList)`**

   El parámetro `prodList` es opcional y sirve para indicar la lista de producciones que se desea ejecutar. Si no se explicita, se ejecutarán todas las producciones previamente definidas en el generador[51].

   Antes de definir una generación, `gen` verificará la ausencia de colisiones entre sus producciones. Si se da alguna colisión, se lanzará la excepción correspondiente.

   Dos producciones colisionan si se cumple cualquiera de las siguientes condiciones:

   a) Sus `oFiles` coinciden, pero no sus `iFiles`.

   b) Sus `iFiles` y sus `oFiles` coinciden, y la lista de sustituciones resultante de concatenar sus `subLists` presenta colisiones.

4.2.3. Implementación de los operadores para la combinación de generadores

Los operadores de ETL están implementados con los siguientes métodos de la clase *Generator*:

a. **Secuencia**. El intérprete de Ruby ejecutará las generaciones según el orden en que estén escritas. Es decir, la combinación de generadores será por defecto secuencial.

b. **Suma.** Para la suma de generadores, *Generator* dispone del operador + y del método `add`:

   - **`+(generator)`**
   - **`add(generator)`**

   Las dos modalidades son equivalentes. Por ejemplo, la suma de tres generadores `G1`, `G2` y `G3` podría expresarse como `G1+G2+G3` ó `G1.add(G2.add(G3))`.

   Para permitir la suma de generadores que comparten el nombre de alguna sustitución o de alguna producción, el generador resultado de + ó `add` contendrá

[51] No dar valor a `prodList` es equivalente a escribir `prodList = 'all'`

las sustituciones y las producciones de los operandos renombradas automáticamente, anteponiéndoles el nombre del generador del que provienen.

+ y `add` verificarán la ausencia de colisiones en el generador resultado. Si se da alguna colisión, se lanzará la excepción correspondiente.

c. **Superposición.** Para la superposición de generadores, *Generator* dispone del operador << y del método `sup!`:

- **`<<(generator)`**
- **`sup!(generator)`**

Las dos modalidades son equivalentes. Por ejemplo, la superposición de tres generadores `G1`, `G2` y `G3` podría expresarse como `G1<<G2<<G3` ó `G1.sup!(G2.sup!(G3))`.

Mientras que la suma no modifica los operandos, la superposición sí, pues se realiza sobre el primer operando. Siguiendo la notación estándar de Ruby, `sup!` lleva el sufijo `!` para alertar de que es un método que modifica sus parámetros.

`<<` y `sup!` verificarán la ausencia de colisiones en el generador resultado. Si se da alguna colisión, se lanzará la excepción correspondiente.

La figura 4.13 resume la implementación de ETL, que puede consultarse en el CDROM anexo a esta tesis. Internamente, *Generator*:

- dispone de los métodos auxiliares `prod?`[52], `gen?`, `add?` y `sup?` para la detección de colisiones.

- posee dos tablas, llamadas `subHash` y `prodHash`, donde se almacenan las sustituciones y las producciones, que son objetos de clase `Substitution` y `Production`. Estas clases cuentan con métodos `crash?` para evaluar la existencia de colisiones.

- utiliza la clase `Error` para la gestión de errores.

[52] Siguiendo la notación estándar de Ruby, los métodos auxiliares `prod?`, `gen?`, `add?`, `sup?` y `crash?` llevan el sufijo `?` para indicar que devuelven booleanos.

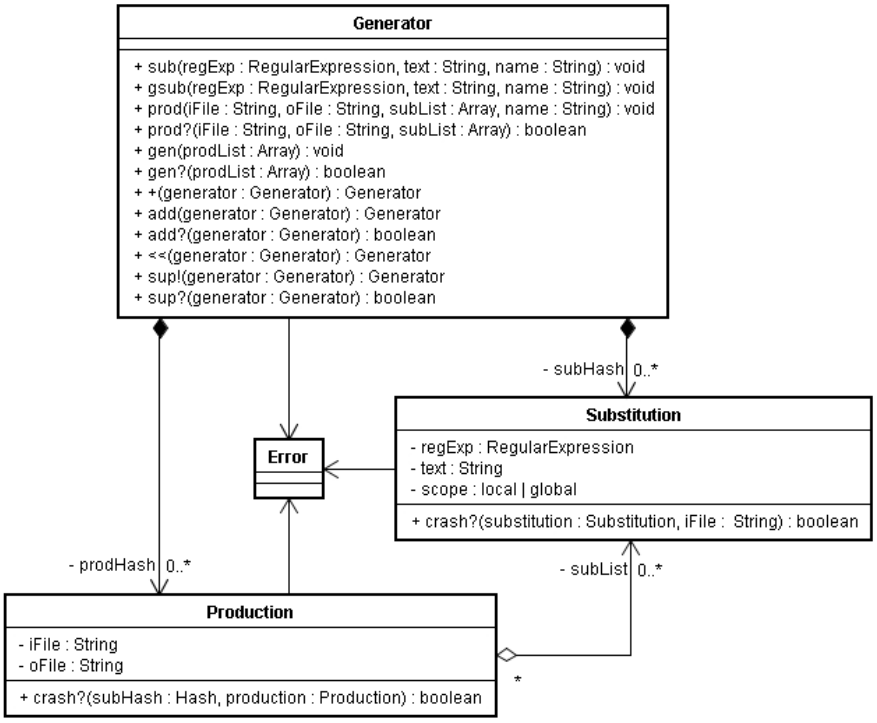


Figura 4.13. Diseño detallado de ETL en Ruby.

4.2.3. Ejemplo

A continuación, se codifica la flexibilización de la familia "programas de reciclaje de basura" planteada en la sección 4.1.4.

4.2.3.1. Definición de la interfaz para parametrizar la flexibilización

La figura 4.14 representa el analizador que obtiene de la especificación XML de un programa de reciclaje el valor de las siguientes variables:

- `out_dir`. Cadena de caracteres que guarda el directorio donde se situará el programa objeto.

- `Trashes`. Tabla que almacena los pares *tipo de basura – precio de reventa.*

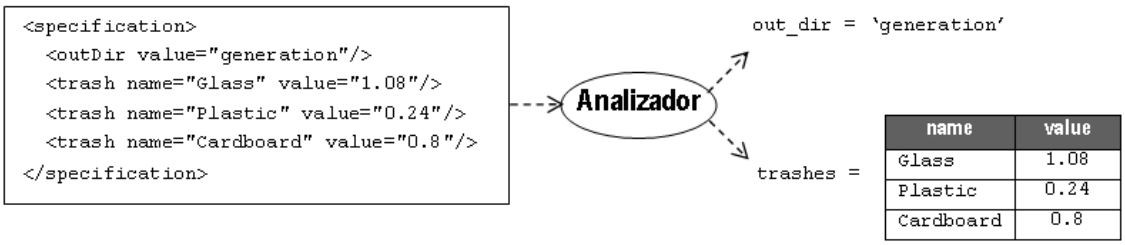


Figura 4.14. Ejemplo de análisis.

La figura 4.15 es el analizador escrito en Ruby, que aprovecha la librería REXML para la manipulación de documentos XML.

```
require 'rexml/document'

def readSpecification
 doc = REXML::Document.new(File.open("specification.xml"))

 out_dir = ''
 doc.root.each_element("outDir") { |e|
  out_dir = e.attributes["value"]
 }

 trashes = {}
 doc.root.each_element("trash") { |e|
  trashes.store(e.attributes["name"],  e.attributes["value"])
 }

 return [out_dir, trashes]
end
```

Figura 4.15. Analizador.

4.2.3.2. Implementación de la flexibilización

Las figuras 4.16 y 4.17 son la codificación de los generadores *RecycleGen* y *TestGen*. Como puede observarse, es necesario importar la librería *ETL*, contenida en la carpeta `ETL` del cd adjunto a esta tesis:

```
require '..\ETL\ETL'
```

y que los generadores hereden de la clase *Generator* :

- `RecycleGen < Generator`
- `TestGen < Generator`

Las sustituciones se han coloreado para facilitar el reconocimiento del código del ejemplar que manipulan.

```
require '..\ETL\ETL'

class RecycleGen < Generator

def initialize(out_dir, trashes)

 # Production 1
 prod(EXEMPLAR_DIR + '\Trash.java', out_dir+'\Trash.java', [], '1')

 # Productions 2
 trashes.each { |kind, value|
  gsub(/Paper/, kind, kind)

  sub(/\d+.\d\d/,
      value,
      "#{kind}_value")

  prod("#{EXEMPLAR_DIR}" + "\\Paper.java",
       "#{out_dir}\\#{kind}.java",
```

```
        [kind, "#{kind}_value"],
        "2_#{kind}")
 }

 # Production 3
 sub(/ArrayList .+ Bin(,ArrayList .+Bin)*/x,
     trashes.keys.collect {|kind|
       "ArrayList " + kind + "Bin"
     }.join(','),
     'argRecycle')

 sub(/(if.+;\s*)+/,
     trashes.keys.collect {|kind|
       "if(t instanceof #{kind}) " + "#{kind}Bin.add(t);\n"
     }.join,
     'ifRecycle')

 prod("#{EXEMPLAR_DIR}" + "\\Recycle.java",
      "#{out_dir}\\Recycle.java",
      ['argRecycle', 'ifRecycle'] ,
      '3')
end

end #RecycleGen
```

Figura 4.16. Generador RecycleGen.

```
require '..\ETL\ETL'

class TestGen < Generator
 def initialize(out_dir, trashes)

  gsub(/aluminumBin, paperBin/,
       trashes.keys.collect {|kind|
         "#{kind.downcase}Bin"
       }.join(', '))

  sub(/(.+Bin( = new ArrayList\(\);\n))+/,
      trashes.keys.collect {|kind|
        "#{kind.downcase}Bin \\2"
      }.join)

  sub(/2/, trashes.length.to_s, 'numberOfTrashes')

  sub(/
       (
         \s*case\s*\d+\s*:\s*
         bin.add\(new\s*\w+\(Math\.random\(\)\s*\*\s*100\)\);\s*
         break;\s*
       )+
     /xm,
     caseTemplate(trashes)
  )

  sub(/(\s*Trash.sumValue\(\w+Bin.+\)\s*(\+)?\s*)+/,
      sumValueTemplate(trashes))

  prod("#{EXEMPLAR_DIR}\\RecycleTest.java", "#{out_dir}\\RecycleTest.java", 'all', '4'))
 end

 def caseTemplate(trashes)
  code = ''
  index = 0
  trashes.each_key {|kind|
   code += "case #{index}:
    bin.add(new #{kind}(Math.random() * 100));
    break;\n"
   index += 1
  }
  return code
```

```
 end

 def sumValueTemplate(trashes)
  code = ''
  index = 0
  trashes.each_key {|kind|
   code += " + " if index > 0
   code += "Trash.sumValue\(#{kind.downcase}Bin.iterator\(\)\)"
   code += "\n"
   index += 1
  }
  return code
 end
end #TestGen
```

Figura 4.17. Generador TestGen.

Como se señaló en la sección 4.1.4.2, el generador *RecycleGen* debe contener tantas producciones 2 como tipos de basura se deseen separar. Para expresar las producciones 2 y sus sustituciones de forma general, se ha utilizado un iterador[53]. El siguiente fragmento de código de la figura 4.16:

```
trashes.each { |kind, value|
  gsub(/Paper/, kind, kind)

  sub(/\d+.\d\d/,
      value,
      "#{kind}_value")

  prod("#{EXEMPLAR_DIR}" + "\\Paper.java",
       "#{out_dir}\\#{kind}.java",
       [kind, "#{kind}_value"],
       "2_#{kind}")
 }
```

recorre la tabla `trashes`, recuperando en cada iteración los campos `kind` y `value`. Con estos campos:

i. Se define una sustitución global, cuyo nombre es el valor de `kind` y que modifica el código variable marcado en □ en la figura 4.5.

ii. Se define una sustitución local, cuyo nombre es el valor de `kind` más el sufijo `_value` y que modifica el código variable marcado en □ en la figura 4.5.

iii. Se define una producción, cuyo nombre es `2_` más el valor de `kind`, que incluye las sustituciones definidas en i e ii.

En las sustituciones O, O, O y O se utilizan los siguientes métodos estándares de Ruby:

[53] Con el propósito de evitar errores en la escritura de bucles, Ruby ofrece iteradores para todos sus contenedores predefinidos.

- `keys`: proporciona una lista formada por las claves de una tabla. En el ejemplo de la figura 4.11, `trashes.keys` devolvería *[Glass, Plastic, Cardboard]*.

- `collect`: es un iterador que recorre una lista, aplicando a cada elemento un código asociado, delimitado entre llaves. En el ejemplo de la figura 4.11, la ejecución de

```
trashes.keys.collect {|kind|
  "ArrayList " + kind + "Bin"
}
```

  convertiría la lista *[Glass, Plastic, Cardboard]* en *[ArrayList GlassBin, ArrayList PlasticBin, ArrayList CardboardBin]*.

- `join`: transforma una lista en una cadena de caracteres concatenando sus elementos. Opcionalmente, puede incluir una cadena de caracteres que intercalará entre los elementos de la lista. En el ejemplo de la figura 4.11, la ejecución de

```
trashes.keys.collect {|kind|
  "ArrayList " + kind + "Bin"
}.join(',')
```

  producirá la cadena

```
"ArrayList GlassBin, ArrayList PlasticBin, ArrayList CardboardBin"
```

Finalmente, la figura 4.18 muestra la ejecución del analizador de la figura 4.15 mediante la sentencia

```
out_dir, trashes = readSpecification
```

y la suma los generadores *RecycleGen* y *TestGen*, ejecutando después la generación del resultado.

```
require '..\ETL\ETL'

EXEMPLAR_DIR = 'exemplar'

out_dir, trashes = readSpecification

( RecycleGen.new(out_dir, trashes) +  TestGen.new(out_dir, trashes) ).gen
```

Figura 4.18. Configuración y ejecución del compilador.

5

Estudio comparativo de metodologías y técnicas de desarrollo de familias de productos

The history of programming is an exercise in hierarchical abstraction. In each generation, language designers produce constructs for lessons learned in the pervious generation, and then architects use them to build more complex and powerful abstractions.

J. Smith, D. Stotts. *Elemental Design Patterns: A Link Between Architecture and Object Semantics.*

El presente capítulo muestra, a través de dos ejemplos, distintas maneras de construir una familia de productos.

En la sección 5.1 se subraya la necesidad de que las infraestructuras que soportan la obtención automática de los productos de una familia dispongan de ocultación (sección 5.1.1). Así mismo, se analiza la trascendencia de que dichas infraestructuras sean intérpretes (sección 5.1.2.1) o compiladores (secciones 5.1.2.2 y 5.1.2.3). La sección 5.1.2.2 muestra los inconvenientes de que un compilador genere código desde cero, escribiendo sobre un flujo de texto la conversión del programa fuente. La sección 5.1.2.3 presenta una generación más cómoda aplicando EDD y ETL.

La sección 5.2 muestra las diferencias metodológicas entre la GP (sección 5.2.1) y EDD (sección 5.2.2), e indica cómo aplicar las técnicas comunes de generalización de código a la flexibilización de un ejemplar y cuáles son sus limitaciones (sección 5.2.3).

Finalmente, la sección 5.3 resume las conclusiones de este capítulo.

5.1. Ejemplo 1: "Desarrollo de diccionarios en Java"

El ejemplo que se plantea a continuación se inspira en el supuesto práctico formulado por J. C. Cleaveland en capítulo 1 de [Cle01].

Muchos IDEs y procesadores de textos disponen de un diccionario para reconocer las palabras de un lenguaje. Los IDEs aprovechan los diccionarios para resaltar las palabras reservadas de un lenguaje de programación, y agilizar así la edición de los programas. Los procesadores de texto suelen resaltar las palabras ausentes en el diccionario para facilitar la corrección de los documentos.

Supóngase que un fabricante de procesadores de textos ha desarrollado, para la revisión de documentos en castellano, el diccionario de la figura 5.1. Como puede apreciarse, el diccionario es una clase escrita en Java que contiene el método `isWord` para averiguar si una palabra pertenece al castellano.

```
public class ReducedSpanishDictionary {

        public boolean isWord(String word) {
                if ((word == "hombre") ||
                    (word == "perro") ||
                    (word == "colegio") ||
                    (word == "vaca"))
                    // etc.
                        return true;
                else
                        return false;
        }

}
```

Figura 5.1. Versión reducida de un diccionario para la revisión ortográfica de documentos en castellano.

Posteriormente, el fabricante detecta la necesidad de disponer de diccionarios para las versiones de cada país de sus procesadores de textos y comienza a cuestionarse la rentabilidad de desarrollar diccionarios de lenguajes de programación que puedan integrarse en los IDEs de otros fabricantes.

Tras el análisis de la potencial familia de productos, se identifican los siguientes requisitos:

- **Requisitos fijos**: los diccionarios serán clases Java con un método `isWord` para verificar la pertenencia de una palabra al lenguaje correspondiente.

- **Requisitos variables**: cada diccionario contendrá un léxico particular que habrá que especificar.

A continuación se examinan distintas maneras de construir una infraestructura que soporte la obtención automática de diccionarios.

5.1.1. Desarrollo de una infraestructura sin ocultación

La infraestructura puede construirse flexibilizando el programa de la figura 5.1. Para ello, se localizará en este ejemplar el único grupo de requisitos variables: el léxico de cada diccionario. La zona del ejemplar que debe flexibilizarse aparece de color amarillo en la figura 5.1 y consiste en una expresión condicional sobre un grupo de literales. El medio que habitualmente ofrecen los lenguajes de programación para generalizar grupos de literales son las **variables**. Introduciendo los valores "hombre", "perro"… en un vector, al que llamaremos `dictionary`, el método `isWord` podrá expresarse de forma general, como aparece en la figura 5.2.

```
import java.util.*;

public class WhiteBoxGeneralizedDictionary {

        private String[] dictionary;

        WhiteBoxGeneralizedDictionary (Language language) {
                dictionary = language.getWords();
        }

        public boolean isWord(String word) {
                for (int i=0; i<dictionary.length; i++) {
                        if (word.equals(dictionary[i]))
                                return true;
                }
                return false;
        }
}
```

Figura 5.2. Flexibilización de la figura 5.1 sin ocultación.

La especificación de un diccionario concreto se conseguirá particularizando el vector `dictionary` a través del constructor de la clase `WhiteBoxGeneralizedDictionary`. Puesto que Java es un lenguaje "fuertemente orientado a objetos", que incentiva que los componentes fundamentales de un programa sean clases, las especificaciones también deberían ser clases. Para que el constructor de `WhiteBoxGeneralizedDictionary` pueda recibir como parámetro de entrada una clase con el léxico particular de un lenguaje, el tipo del parámetro debe ser variable y abarcar el rango de todas las clases que

implementan diccionarios. Los medios que suelen proporcionar los lenguajes de programación para flexibilizar los tipos son la herencia de clases y la genericidad. En este ejemplo se utilizará la **herencia de clases**, en la sección 5.2 se usará genericidad. El parámetro de entrada del constructor será la interfaz o tipo abstracto *Language* de la figura 5.3. Como indica la figura 5.4, las clases que especifiquen diccionarios deberán implementar la interfaz *Language*. Gracias al **enlace dinámico**, en la ejecución de la sentencia

```
dictionary = language.getWords();
```

se seleccionará automáticamente el método *getWords* de la clase adecuada. Las figuras 5.5 y 5.6 son dos ejemplos de especificación de un diccionario de inglés y un diccionario del lenguaje de programación Modula-2 respectivamente.

```
public interface Language {
  public String[] getWords();
}
```

Figura 5.3. Interfaz Language.

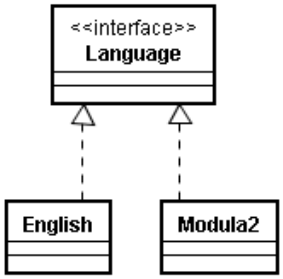


Figura 5.4. La especificación de cada diccionario implementará Language.

```
public class English implements Language {

  public String[] words = {
        "man",
        "dog",
        "school",
        "cow"
        // etc.
  };

  public String[] getWords() {
    return words;
  }
}
```

Figura 5.5. Especificación de un diccionario de inglés para la infraestructura de la figura 5.2.

```
public class Modula2 implements Language {

  public String[] words = {
        "FOR",
        "IF",
        "IMPORT",
        "BEGIN"
        // etc.
  };

  public String[] getWords() {
    return words;
  }
}
```

Figura 5.6. Especificación de un diccionario de Modula-2 para la infraestructura de la figura 5.2.

Este ejemplo permite apreciar los siguientes inconvenientes típicos de las infraestructuras sin ocultación:

- La obligación de introducir detalles de implementación de la infraestructura en la especificación de los productos hace que las especificaciones sean poco legibles e innecesariamente prolijas (código en amarillo de las figuras 5.5 y 5.6).

- La curva de aprendizaje de la infraestructura es considerable. En este caso, será imprescindible saber que la especificación de un producto se consigue implementando la interfaz `Language`.

- Se introduce un fuerte acoplamiento entre las especificaciones y la implementación de la infraestructura, que probablemente obligará a reescribir las especificaciones si se modifica esta última.

5.1.2. Desarrollo de una infraestructura con ocultación

Para evitar los inconvenientes anteriores, puede dotarse de ocultación a la infraestructura. La figura 5.7 es la **gramática** de un DSL que permitirá especificar diccionarios de forma abstracta, excluyendo los detalles de implementación de la infraestructura. Las figuras 5.8 y 5.9 son dos ejemplos de especificaciones DSL análogas a las figuras 5.5 y 5.6.

En la sección 5.2.2 se estudiará un ejemplo más complicado y se expondrá la obtención sistemática de una gramática independiente del contexto a partir del modelo de un dominio.

```
Diccionario ::= Lexico;
Lexico ::= {alpha}[54];
```

Figura 5.7. Gramática de un DSL para especificar diccionarios.

```
Man
dog
school
cow
…
```

Figura 5.8. Especificación, según la gramática de la figura 5.7, de un diccionario de inglés.

```
FOR
IF
IMPORT
BEGIN
…
```

Figura 5.9. Especificación, según la gramática de la figura 5.7, de un diccionario de Modula-2.

Las siguientes secciones muestran la trascendencia de implementar la infraestructura como un intérprete (5.1.2.1) o como un compilador (5.1.2.2 y 5.1.2.3).

5.1.2.1. Infraestructura implementada como un intérprete

Las figuras 5.10 y 5.11 son, respectivamente, el diseño arquitectónico y la codificación de la infraestructura construida como intérprete.

[54] En la notación de *ProGrammar* [PGram06] `alpha` es equivalente a la expresión regular `[a-zA-Z]+`.

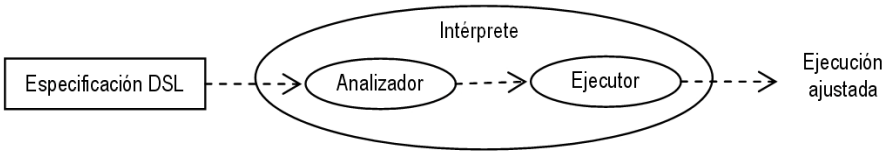


Figura 5.10. Arquitectura de un intérprete de diccionarios.

El método `analyzeLanguage` de la figura 5.11 analiza las especificaciones DSL de entrada, convirtiéndolas en representaciones internas (los vectores `dictionary`) fácilmente manipulables por el "ejecutor".

```
import java.util.*;
import java.io.*;

public class BlackBoxGeneralizedDictionary {

        private Vector dictionary = new Vector();
        private String languageFile;

        BlackBoxGeneralizedDictionary(String languageFile) {
                this.languageFile = languageFile;
                analyzeLanguage();
        }

        ////////////////////////////////////////////////////
        // Analyzer
        ////////////////////////////////////////////////////
        public void analyzeLanguage() {
                try {
                        BufferedReader f = new BufferedReader(
                                new FileReader(languageFile));
                        String word = null;
                        while ((word = f.readLine()) != null) {
                                dictionary.addElement(word);
                        }
                } catch (Exception e) {
                        System.err.println("Unable to read the language file");
                }
        }

        ////////////////////////////////////////////////////
        // Executor
        ////////////////////////////////////////////////////
        public boolean isWord(String word) {
                for (int i=0; i<dictionary.size(); i++) {
                        String wordAux = (String) dictionary.elementAt(i);
                        if (word.equals(wordAux))
                                return true;
                }
                   return false;
        }
}
```

Figura 5.11. Codificación de un intérprete de diccionarios.

Como se indicó en la sección 3.2.3, mientras que la variabilidad interna de un producto normalmente debe poder ajustarse en tiempo de ejecución, no suele ocurrir así con la variabilidad entre los productos de una familia. Lamentablemente, cuando la infraestructura es un intérprete, la variabilidad entre los productos se gestiona en tiempo de ejecución, lo que puede ralentizar innecesariamente los productos. En nuestro ejemplo,

cada vez que se instancie un diccionario se invertirá un tiempo inútil en leer y cargar en memoria la especificación DSL correspondiente. Esto puede evitarse haciendo que la infrestructura sea un compilador que genere diccionarios específicos. Las secciones 5.1.2.2 y 5.1.2.3 presentan dos enfoques alternativos para la generación: la escritura desde cero sobre un flujo de texto y la flexibilización del ejemplar con ETL.

5.1.2.2. Generación de productos desde cero, escribiendo sobre un flujo de texto

Las figuras 5.12 y 5.13 son respectivamente el diseño arquitectónico y la codificación de un compilador que crea diccionarios desde cero, escribiendo la conversión de las especificaciones DSL sobre un flujo de texto.

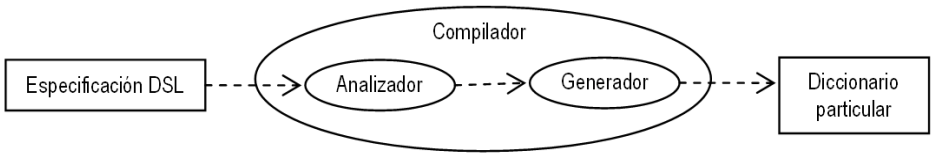


Figura 5.12. Arquitectura de un compilador de diccionarios. Generación desde cero, escribiendo sobre un flujo de texto.

```
import java.io.*;
import java.util.*;

class ClassicTransformer {
  private Vector dictionary = new Vector();
  private String languageFile;
  private String language;

  ClassicTransformer(String languageFile) {
    this.languageFile = languageFile;
    StringTokenizer st =
      new StringTokenizer(languageFile, ".");
    language = st.nextToken();
    analyzeLanguage();
    generateDictionary();
  }

  ////////////////////////////////////////////////////
  // Analyzer
  ////////////////////////////////////////////////////
  public void analyzeLanguage() {
    try {
      BufferedReader f = new BufferedReader(
        new FileReader(languageFile));
      String word = null;
      while ((word = f.readLine()) != null) {
        dictionary.addElement(word);
      }
    } catch (Exception e) {
      System.err.println(
        "Unable to read the dictionary specification file");
    }
  }

  ////////////////////////////////////////////////////
  // Generator
```

```
  ////////////////////////////////////////////////////
  public void generateDictionary() {
    try {
      BufferedWriter f = new BufferedWriter(
        new FileWriter(language+"Dictionary.java"));
      f.write("public class "+language+"Dictionary {\n");
      f.write("  public boolean isWord(String word) {\n");
      f.write("    if (\n");
      for (int i=0; i<dictionary.size()-1; ++i) {
        f.write("      (word == \""+dictionary.elementAt(i)+"\") ||\n");
      }
      f.write("      (word == \""+dictionary.elementAt(dictionary.size()-1)+"\")\n");
          f.write("    )\n");
      f.write("      return true;\n");
      f.write("    else\n");
      f.write("      return false;\n");
      f.write("  }\n");
      f.write("}\n");
      f.close();
    } catch (Exception e) {
      System.err.println("Unable to generateDictionary file Static"+
        language+"Dictionary.java");
    }
  }
}
```

Figura 5.13. Codificación de un compilador de diccionarios. Generación desde cero, escribiendo sobre un flujo de texto.

Las figuras 5.14 y 5.15 son los diccionarios generados a partir de las especificaciones de las figuras 5.8 y 5.9 respectivamente.

```
public class EnglishDictionary {
  public boolean isWord(String word) {
    if (
      (word == "man") ||
      (word == "dog") ||
      (word == "school") ||
      (word == "cow")
    )
      return true;
    else
      return false;
  }
}
```

Figura 5.14. Diccionario de inglés generado por la infraestructura de la figura 5.13 a partir de la especificación de la figura 5.8.

```
public class Modula2Dictionary {
  public boolean isWord(String word) {
    if (
      (word == "IF") ||
      (word == "FOR") ||
      (word == "IMPORT") ||
      (word == "BEGIN")
    )
      return true;
    else
      return false;
  }
}
```

Figura 5.15. Diccionario deModula-2 generado por la infraestructura de la figura 5.13 a partir de la especificación de la figura 5.9.

5.1.2.3. Flexibilización de un ejemplar con ETL

Dada la proximidad entre los productos del generador (compárense las figuras 5.14 y 5.15) y la lejanía entre las especificaciones DSL y los productos finales (compárense las figuras 5.8 y 5.14), esta tesis propone que los productos, en lugar de generarse desde su especificación, se obtengan adaptando un ejemplar del dominio.

Las figuras 5.16, 5.17 y 5.18 son, respectivamente, el diseño arquitectónico, el diseño detallado y la codificación de una flexibilización con ETL del ejemplar de la figura 5.1. Como ilustra la figura 5.17, el generador ETL (la clase *DictionaryGenerator*) consta de dos sustituciones, que ajustan el nombre de la clase diccionario (en rosa) y la expresión condicional de la sentencia de selección (en azul), y de una producción que aplica las sustituciones sobre el ejemplar para producir las clases Java resultado.

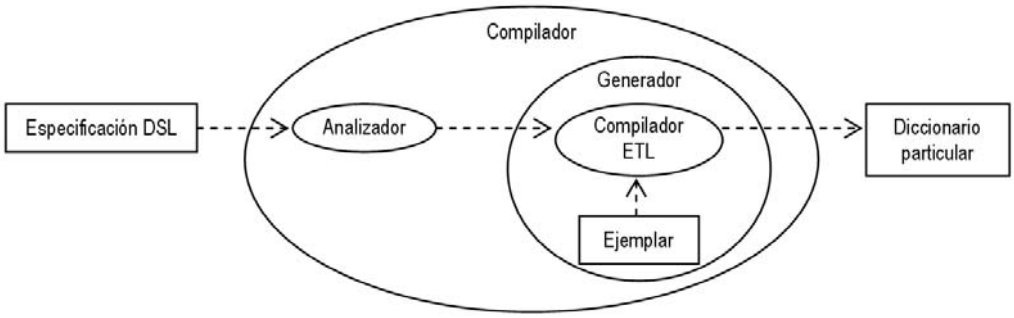


Figura 5.16. Arquitectura de un compilador de diccionarios. Generación con ETL por analogía.

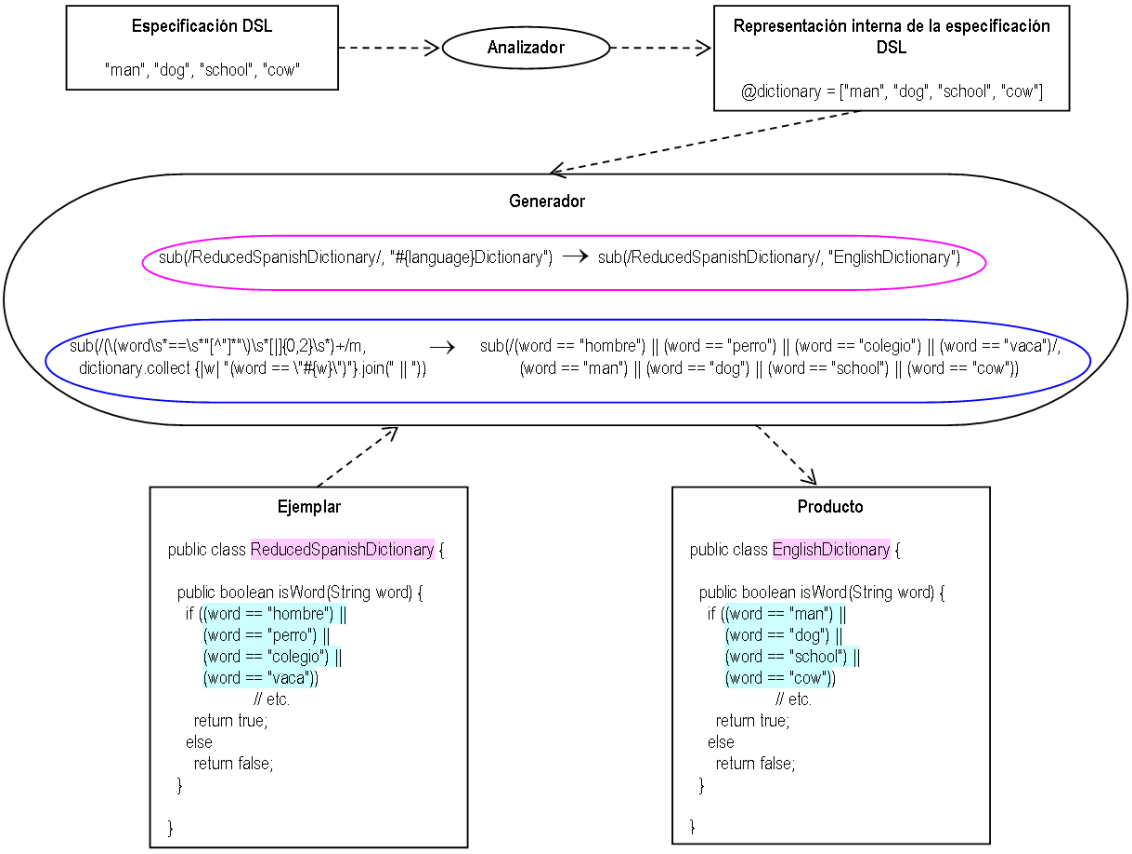


Figura 5.17. Diseño detallado de un compilador de diccionarios. Generación con ETL por analogía.

```
require '..\..\..\..\ETL\ETL'

##########################################################
# Generator
##########################################################
```

```
class DictionaryGenerator < Generator
  def initialize(language, dictionary)
    sub(/ReducedSpanishDictionary/, "#{language}Dictionary")
    sub(/(\(word\s*==\s*"[^"]*"\)\s*[|]{0,2}\s*)+/m,
      dictionary.collect {|w| "(word == \"#{w}\")"}.join(" || "))
    prod("ReducedSpanishDictionary.java", "#{language}Dictionary.java")
  end
end

unless ARGV[0]
  print "You should specify the dictionary file"
  exit
end
languageFile = ARGV[0]

##########################################################
# "Analyzer"
##########################################################
eval "@dictionary = [#{File.open("#{languageFile}").read}]"

##########################################################
# Running the generator
##########################################################
DictionaryGenerator.new(languageFile.split(/\./)[0], @dictionary).gen
```

Figura 5.18. Codificación de un compilador de diccionarios. Generación con ETL por analogía.

En la figura 5.18, para simplificar el análisis de las especificaciones, se ha utilizado un **DSL interno**. A partir de la gramática de la figura 5.7, se ha escrito la **gramática concreta** de la figura 5.19. Con la nueva gramática, se tendrá la sensación de escribir código DSL, cuando en realidad estará escribiendo código Ruby como el de la figura 5.20. Así, el análisis se reduce a la ejecución de la sentencia

```
eval "@dictionary = [#{File.open("#{languageFile}").read}]"
```

La figura 5.21 muestra, paso a paso, cómo será ejecutada la sentencia anterior por el intérprete de Ruby, suponiendo que la variable `languageFile` valga "`English.txt`" (figura 5.20).

```
Diccionario ::= Lexico;
Lexico ::= {"\""alpha"\"", ","};
```

Figura 5.19. Gramática, equivalente a la de la figura 5.7, de un DSL interno a Ruby

```
"man", "dog", "school", "cow"
```

Figura 5.20. Especificación, según la gramática de la figura 5.18, de un diccionario de inglés.

5.1.2.4. Comparación entre la generación de productos con ETL y la generación desde cero, escribiendo sobre un flujo de texto

Respecto a la generación desde cero, la generación ETL aporta:

- Concisión en la escritura de los generadores, pues se evita la descripción redundante del código fijo del dominio, ya incluido en el ejemplar. Compárese la

clase `DictionaryGenerator` de la figura 5.18 con el método `generateDictionary` de la figura 5.13.

- Desacoplamiento entre el código fijo y el código variable, que se localizan respectivamente en el ejemplar y en los generadores.

		Aclaraciones
1	`eval "@dictionary = [#{File.open("#{languageFile}").read}]"`	En Ruby, `#{expresión}` dentro de un `string`, indica la inclusión del resultado de evaluar la `expresión` en el `string`. Como la variable `languageFile` vale "`English.txt`", el intérprete sustituirá `#{languageFile}` por `English.txt`.
2	`eval "@dictionary = [#{File.open("English.txt").read}]"`	Se lee el archivo English.txt y se inserta su contenido (figura 5.19) en el `string`.
3	`eval "@dictionary = ["man", "dog", "school", "cow"]"`	El procedimiento estándar `eval string` provoca que el intérprete de Ruby ejecute el `string`.
4	Como resultado de la ejecución de `eval`, existe en memoria una variable, llamada `@dictionary`, cuyo contenido es **`["man", "dog", "school", "cow"]`**	

Figura 5.21. Ejecución, paso a paso, del análisis interno realizado por el intérprete de Ruby de la especificación de la figura 5.20.

5.2. Ejemplo 2: "Desarrollo de una familia de listas para C++"

En el capítulo 12 de [CE00], K. Czarnecki y U. Eisenecker proponen, como ejemplo para aclarar su metodología de GP[55], el desarrollo de un modelo generativo que facilite la construcción de listas en C++. En esta sección se aprovechará este ejemplo para comparar

[55] La metodología propuesta por K. Czarnecki y U. Eisenecker se resumió en la sección 2.2.1.

EDD con la citada metodología. Además, el ejemplo servirá para examinar cómo aplicar las técnicas comunes de generalización de código a la flexibilización de un ejemplar y ver cuáles son sus limitaciones.

5.2.1. Resolución mediante GP

Esta sección resume el capítulo 12 de [CE00].

I. Análisis del dominio

La familia de productos que se desea construir ha de satisfacer los siguientes requisitos, representados en el modelo de la figura 5.22:

- La clase de elementos que las listas pueden almacenar es parametrizable *([ElementType[56]])*.
- Los elementos pueden almacenarse en una lista según las siguientes modalidades (*Ownership*):
    - La lista guarda referencias de sus elementos y, en caso de que la lista se borre, ésta no se responsabiliza de liberar la memoria que ocupan los elementos *(External reference)*.
    - La lista guarda referencias de sus elementos y, en caso de que la lista se borre, ésta se responsabiliza de liberar la memoria que ocupan los elementos *(Owned reference)*.
    - La lista guarda copias de sus elementos y, en caso de que la lista se borre, ésta se responsabiliza de liberar la memoria que ocupan las copias *(Copy)*.
- Para conocer su longitud, una lista puede disponer de un contador *(LengthCounter)*, de tipo configurable *([LengthType])*, que guarde su número de elementos.
- Para facilitar la depuración de productos, una lista puede imprimir en consola cualquier llamada a sus métodos *(Tracing)*.

[56] En la figura 5.22, los corchetes alrededor de una característica *([ElementType], [LengthType])* señalan que la característica admite un rango de valores potencialmente infinito.

Sin contar con la variabilidad que introducen *[ElementType]* y *[LengthType]*, la familia de productos contiene 12 variantes de listas.

II. Diseño arquitectónico del dominio

Se trata de diseñar la arquitectura de una infraestructura que dé soporte al desarrollo automático de los productos de la familia, siguiendo los pasos que se resumen a continuación:

a) Identificar en el modelo del dominio las responsabilidades fundamentales de la línea de productos

Según los autores, el nodo *List* de la figura 5.22 sugiere la responsabilidad de almacenar elementos, *Ownership* las responsabilidades de copiar y liberar elementos, *LengthCounter* la responsabilidad de contar el número de elementos de una lista y *Tracing* la responsabilidad de imprimir mensajes que señalen la ejecución de los métodos de una lista.

Lamentablemente, tal y como se aprecia en la figura 5.23, el paso del diagrama de características a las responsabilidades fundamentales que debe asumir la línea de productos no es trivial:

- Generalmente, la responsabilidad derivada de un nodo incluye a sus nodos hijos (*Ownership*, *LengthCounter*). Sin embargo, no siempre es así (*List*).

- Aunque normalmente la correspondencia *nodo*→*responsabilidad* es una aplicación biyectiva (*List, LengthCounter, Tracing*). Puede interesar, por ejemplo, que a un nodo le correspondan varias responsabilidades (*Ownership*).

- Algunos nodos no sugieren ninguna responsabilidad importante *(ElementType)*.

b) Identificar los componentes comunes de la familia de productos

Para cada responsabilidad, se determinará una categoría de componentes que cubran las alternativas descritas en el modelo del dominio. Por ejemplo, los tipos de pertenencia de un elemento a una lista (*Ownership*) se cubrirán con las categorías de componentes *Copier* y *Destroyer* del siguiente modo: *External reference*→*EmptyCopier* & *EmptyDestroyer; Owned reference*→*EmptyCopier* & *ElementDestroyer; Copy*→*ElementCopier* & *ElementDestroyer.*

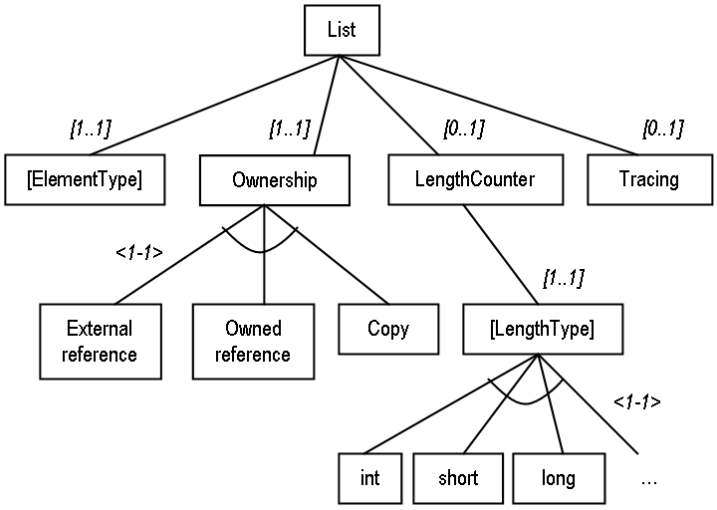


Figura 5.22. Modelo FODA del dominio "listas para C++".

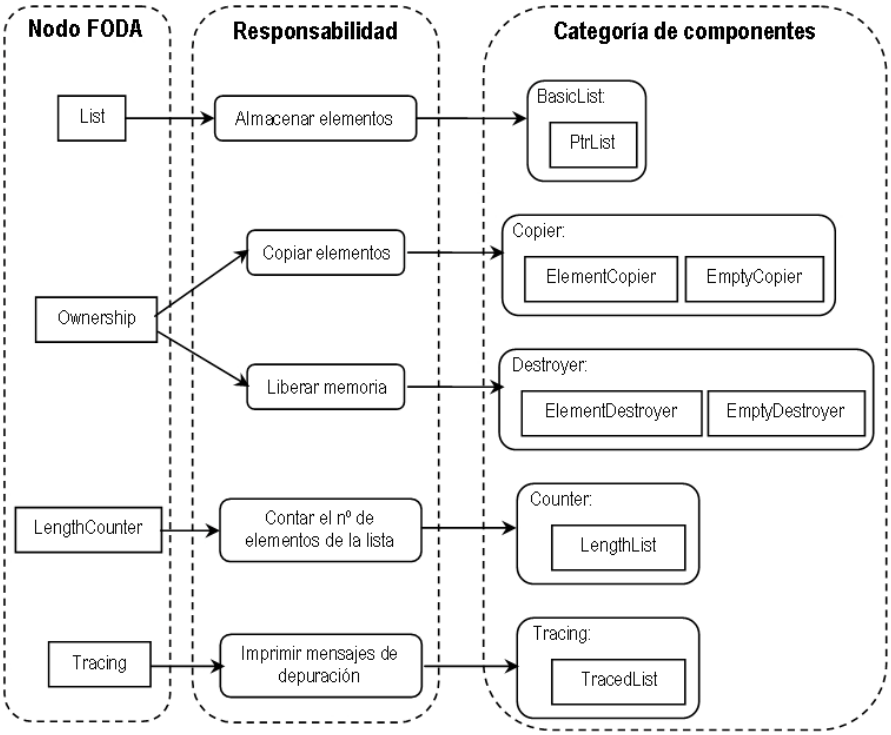


Figura 5.23. Correspondencias entre los nodos del diagrama de la figura 5.22, las responsabilidades que debe asumir la línea de productos y las categorías de componentes que deben desarrollarse.

c) Identificar las dependencias de uso entre las categorías de componentes

Según los autores, las categorías de componentes *Counter* y *Tracing* utilizarán la categoría *BasicList*, que a su vez utilizará a *Copier* y a *Destroyer* (ver figura 5.24).

d) Situar las categorías de componentes en una arquitectura por capas

Teniendo en cuenta que una $capa_N$ utilizará los servicios de la $capa_{N-1}$ y proveerá de servicios a la $capa_{N+1}$, se sitúan las categorías de componentes respetando las dependencias de uso antes identificadas (figura 5.25).

Puesto que no hay dependencia entre *Tracing* y *Counter*, es indiferente situar *Tracing* sobre *Counter* o *Counter* sobre *Tracing*. En la figura 5.25, el trazo discontinuo alrededor de *Tracing* y de *Counter* indica que son capas opcionales.

Según los autores, las categorías de componentes más pequeños y la gestión de los tipos se describirán conjuntamente en un "almacén de configuración" *(configuration repository)* que ocupará la capa inferior de la arquitectura. A continuación, se determinará qué categoría de componentes es la principal. Lamentablemente, los autores no aportan ningún criterio para dicha determinación. En nuestro ejemplo, la categoría principal de componentes es *BasicList*. Como se verá en el punto III, los componentes que ocupan las capas superiores a la categoría principal de componentes se implementarán con herencia, mientras que los componentes de las capas inferiores se implementarán con composición. La variabilidad acerca de los tipos se implementará mediante genericidad, haciendo que los componentes *TracedList, LengthList, PtrList, ElementCopier, EmptyCopier, ElementDestroyer* y *EmptyDestroyer* sean plantillas (*templates*). Para evitar una enumeración exhaustiva de todos los tipos (en el ejemplo, *ElementType* y *LengthType*) cada vez que se define o utiliza una plantilla, toda la información sobre los tipos se aglutinará en el almacén de configuración[57].

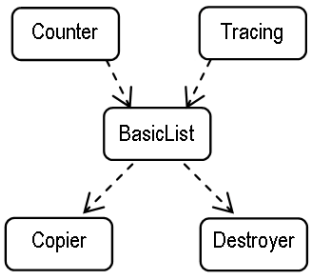


Figura 5.24. Dependencias de uso entre las categorías de componentes.

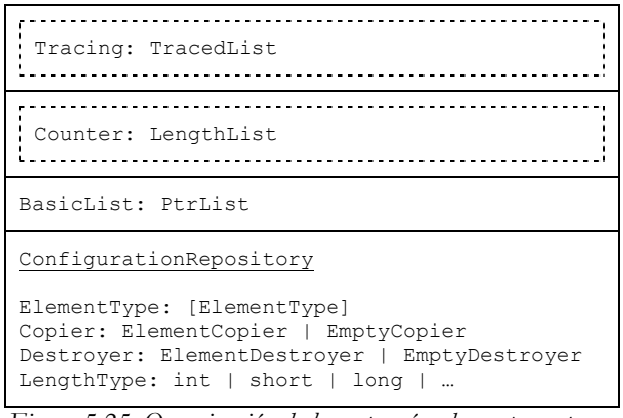


Figura 5.25. Organización de las categorías de componentes en una arquitectura por capas.

[57] Así, en C++, en lugar de tener que escribir:

```
class Componente <Tipo1, Tipo2, ..., TipoN> {...}
```

se escribirá:

```
class Componente <Almacén de configuración> {...}
```

e) Especificar una gramática GenVoca

GenVoca es una propuesta de D. Batory et al. [BCRW98, BO92] para construir generadores de aplicaciones componiendo capas de abstracción orientadas a objetos. Posteriormente, GenVoca ha evolucionado al modelo AHEAD [Bat04, BSR04]. En nuestro ejemplo, la gramática de la figura 5.26 especifica cómo pueden combinarse los componentes de las categorías. Por ejemplo, una lista (*List*) puede obtenerse combinando una *TracedList* con una *OptCounterList* o, simplemente, a partir de una *OptCounterList*, que a su vez puede obtenerse combinando una *LengthList* con una *BasicList* o a partir de una *BasicList*…

```
List: TracedList[OptCounterList] | OptCounterList
OptCounterList: LengthList[BasicList] | BasicList
BasicList: PtrList[Config]
Config:
   ElementType: [ElementType]
   Copier: ElementCopier | EmptyCopier
   Destroyer: ElementDestroyer | EmptyDestroyer
   LengthType: int | short |long | ...
   ReturnType //the final list type
```

Figura 5.26. Gramática GenVoca que especifica las posibles combinaciones entre componentes.

III. Diseño detallado del dominio

Para el diseño detallado y la codificación de los componentes, la GP sugiere aplicar las mismas técnicas que habitualmente se utilizan en la construcción de productos aislados: herencia, composición, genericidad, aspectos…

Concretamente, la metodología expuesta en [CE00] recomienda:

- Gestionar la variabilidad de los tipos mediante genericidad.
- Implementar los componentes de las categorías inferiores a la principal con composición utilizando el patrón de diseño *strategy* [GHJV94] implementado con genericidad en lugar de con herencia de interfaz (en las secciones 5.2.3.3 y 5.2.3.4 se examinan estas dos alternativas).
- Implementar los componentes de las categorías superiores a la principal mediante herencia. Exactamente mediante *herencia parametrizada*, es decir aprovechando la genericidad para que la clase padre sea un parámetro (en las secciones 5.2.3.5, 5.2.3.6 y 5.2.3.7 se justifica este tipo de herencia).

La figura 5.27 muestra el diseño detallado del presente ejemplo.

IV. Codificación de los componentes

La figura 5.28 es el código de los componentes en C++. Se han utilizado distintos colores para distinguir el código que implementa cada requisito del dominio. Como puede apreciarse con la mezcla colores, el código sufre dispersión, lo que dificutará el mantenimiento de la línea de productos.

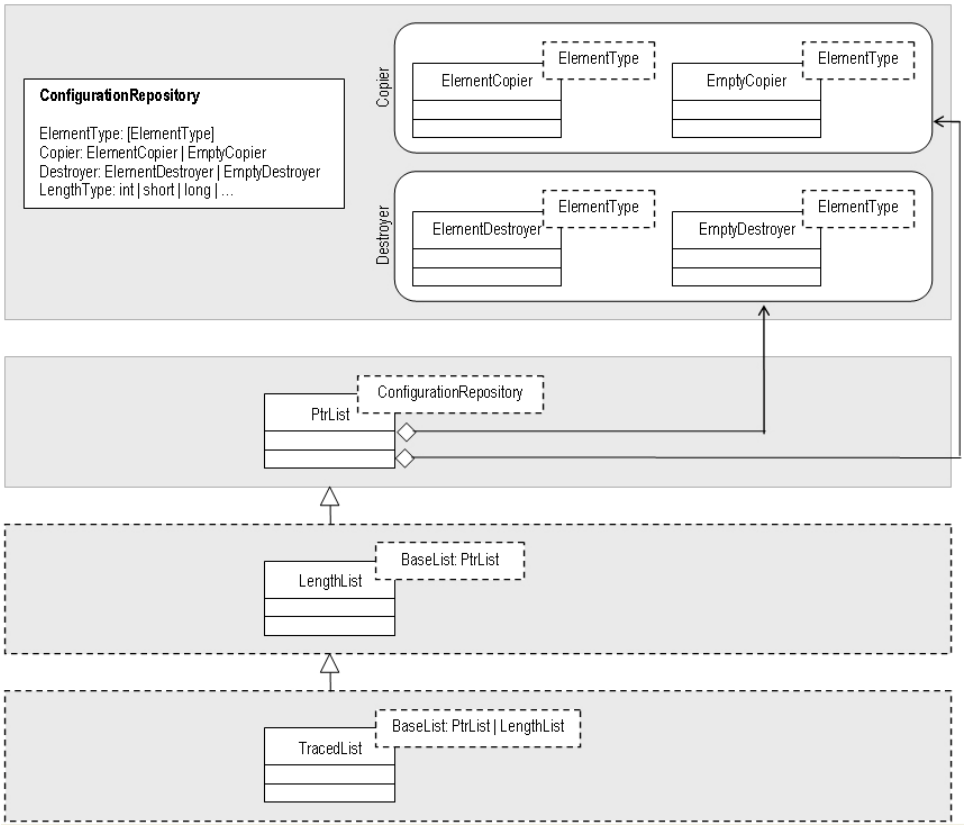


Figura 5.27. Diseño detallado, obtenido con la metodología propuesta en [CE00], de la línea de productos "listas para C++".

```
#include <iostream>
using namespace std;

////////////////////////////////////////////////////////
// TracedList ////////////////////////////////////////
////////////////////////////////////////////////////////

template<class BaseList>
class TracedList: public BaseList
{
public:
        typedef typename BaseList::Config Config;

private:
        typedef typename Config::ElementType ElementType;
        typedef typename Config::ReturnType ReturnType;

public:
        TracedList(ElementType& h, ReturnType *t = 0) :
          BaseList(h, t)
        {}
```

```
        void setHead(ElementType& h)
        {
                cout << "setHead(" << h << ")" << endl;
                BaseList::setHead(h);
        }

        ElementType& head()
        {
                cout << "head()" << endl;
                return BaseList::head();
        }

        void setTail(ReturnType *t)
        {
                cout << "setTail(t)" << endl;
                BaseList::setTail(t);
        }

        ReturnType *tail() const
        {
                cout << "tail()" << endl;
                return BaseList::tail();
        }
};
```

```
//////////////////////////////////////////////////////
// LengthList ////////////////////////////////////////
//////////////////////////////////////////////////////

template<class BaseList>
class LengthList : public BaseList
{
public:
        typedef typename BaseList::Config Config;

private:
        typedef typename Config::ElementType ElementType;
        typedef typename Config::ReturnType ReturnType;
        typedef typename Config::LengthType LengthType;

public:
        LengthList(ElementType& h, ReturnType *t = 0) :
          BaseList(h, t), length (computedLength())
        {}

        void setTail(ReturnType *t)
        {
                BaseList::setTail(t);
                length  = computedLength();
        }

        const LengthType& length() const
        { return length ; }

private:
        LengthType computedLength() const
        { return tail()?tail()->length()+1:1;  }

        LengthType length ;
};
```

```
//////////////////////////////////////////////////////
// Copier ////////////////////////////////////////////
//////////////////////////////////////////////////////

template<class ElementType>
struct ElementCopier
{
        static ElementType* copy(const ElementType& e)
        { return new ElementType(e); }
};

template<class ElementType>
struct EmptyCopier
```

```
{
        static ElementType* copy(ElementType& e)
        { return &e; }
};

//////////////////////////////////////////////////////////
// Destroyer ////////////////////////////////////////////
//////////////////////////////////////////////////////////

template<class ElementType>
struct ElementDestroyer
{
        static void destroy(ElementType *e)
        { delete e; }
};

template<class ElementType>
struct EmptyDestroyer
{
        static void destroy(ElementType *e)
        {}
};

//////////////////////////////////////////////////////////
// PtrList /////////////////////////////////////////////
//////////////////////////////////////////////////////////

template<class Config_>
class PtrList
{
public:
        typedef Config  Config;

private:
        typedef typename Config::ElementType ElementType;
        typedef typename Config::ReturnType ReturnType;
        typedef typename Config::Destroyer Destroyer;
        typedef typename Config::Copier Copier;

public:
        PtrList(ElementType& h, ReturnType *t = 0) :
                head (0), tail (t)
        { setHead(h); }

        ~PtrList()
        { Destroyer::destroy(head_); }

        void setHead(ElementType& h)
        {
                head  = Copier::copy(h);
        }

        ElementType& head()
        { return *head ; }

        void setTail(ReturnType *t)
        { tail_ = t; }

        ReturnType *tail() const
        { return tail ; }

private:
        ElementType* head_;
        ReturnType* tail_;
};
```

Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.28. Codificación de la línea de productos "listas para C++" obtenida con la metodología propuesta en [CE00].

V. Ingeniería de aplicación

Aunque en las páginas 586-590 de [CE00] se indica cómo ocultar los detalles de implementación de la línea de productos con un DSL, por brevedad se obviará este paso. Sin considerar la existencia del citado DSL, la especificación de un producto se realizaría describiendo el almacén de configuración correspondiente. Por ejemplo, la figura 5.29 es un almacén que describe una lista que guarda elementos de clase *MyClass*, con pertenencia de tipo *Copy*, que dispone de un contador de tipo *short* para conocer la longitud de la lista y que imprime en la consola mensajes que señalan la ejecución de los métodos de la lista. La figura 5.30 muestra la combinación de los componentes del espacio de la solución que produce la lista anterior.

```
struct TracedCopyMyClassLenListConfig
{
  typedef MyClass ElementType;
  typedef short LengthType;
  typedef ElementDestroyer<ElementType> Destroyer;
  typedef ElementCopier<ElementType> Copier;

  typedef
    TracedList<
      LengthList<
        PtrList<TracedCopyMyClassLenListConfig>
      >
    > ReturnType;
};
typedef
  TracedCopyMyClassLenListConfig::ReturnType
    TracedCopyMyClassLenList;
```

Figura 5.29. Ejemplo de almacén de configuración.

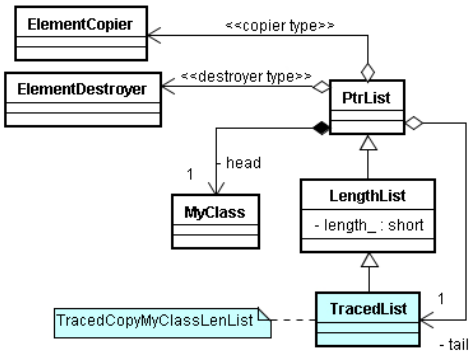


Figura 5.30. Combinación de las clases de la figura 5.28, que satisface la especificación de la figura 5.29.

5.2.2. Resolución mediante EDD y ETL

I. Análisis de la familia de productos

Se supondrá la falta de ejemplares del dominio. El análisis de la familia de productos, al no poder basarse en el análisis de un producto concreto desarrollado con anterioridad, coincidirá con el análisis resumido en el punto I la sección 5.2.1.

II. Construcción de un ejemplar de la familia de productos

Como base para el desarrollo de la familia de productos, se construirá un ejemplar del dominio. El análisis del ejemplar consistirá en la selección de todos los requisitos fijos de la familia, más un conjunto representativo de los requisitos variables.

Según se indicó en la sección 3.2.2.1.1, los nodos fijos de un diagrama FODA se determinan del siguiente modo:

3) Todos los nodos hoja obligatorios (con cardinalidad [1..1]) son fijos.

4) Un nodo, que no sea hoja, es fijo si es obligatorio y todos sus descendientes son fijos.

En el diagrama de la figura 5.22 no hay ningún nodo fijo, ya que los corchetes alrededor de *ElementType* y *LengthType* indican grupos *or* exclusivos de un número potencialmente infinito de valores (figura 5.31).

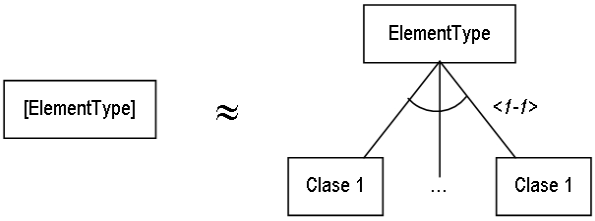


Figura 5.31. Equivalencia entre notaciones FODA.

Siguiendo las pautas descritas en la sección 3.2.1.6, se escogerán, por ejemplo, los requisitos variables que aparecen en negro en la figura 5.32. A continuación, se diseñará, codificará y probará un ejemplar que satisfaga dichos requisitos. La figura 5.33 es la codificación del ejemplar que se utilizará en el resto de la sección. El código en blanco implementa los requisitos fijos del dominio, mientras que el código de color □, □, □ y □ implementa los requisitos variables.

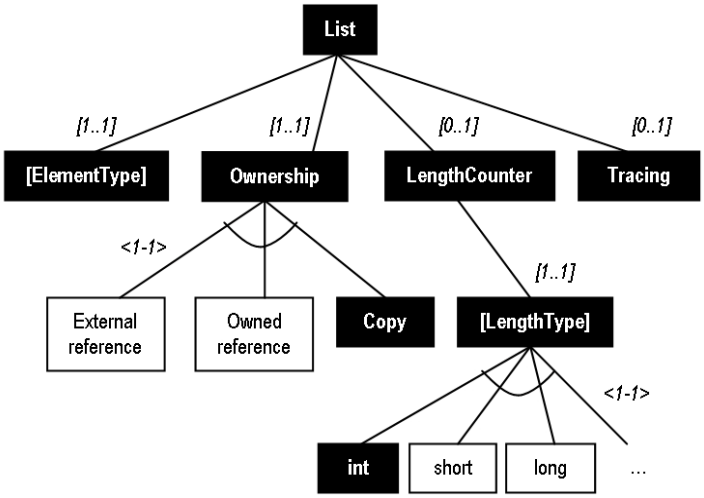


Figura 5.32. Requisitos de la familia "listas para C++" seleccionados para el desarrollo de un ejemplar.

```
class List
{
```

```
private:
        MyClass* head_;
        List* tail_;
        int length_;
public:
        List(MyClass&h, List *t=0):
          head_(0), tail_(t), length_(computedLength())
          { setHead(h); }

        ~List()
        { delete head_; }

        void setHead(MyClass& h)
        {
                cout << "setHead(" << h <<")" << endl;
                head_ = new MyClass(h);
        }

        MyClass& head()
        {
                cout << "head()" << endl;
                return *head_;
        }

        void setTail(List *t)
        {
                cout << "setTail(t)" << endl;
                tail_ = t;
                length_ = computedLength();
        }

        List *tail() const
        {
                cout << "setTail(t)" << endl;
                return tail_;
        }

        const int& length() const
        { return length_; }

private:
        int computedLength() const
        { return tail()?tail()->length()+1:1; }
};
```

Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.33. Ejemplar de la familia "listas para C++".

III. Definición de una interfaz para parametrizar la flexibilización del ejemplar

Aplicando el proceso propuesto en la sección 3.2.2.1.1, el diagrama FODA de la figura 5.22 se traducirá en una gramática independiente del contexto. A continuación, se resumen los pasos que componen la traducción.

Paso 1. Poda de los nodos fijos del diagrama FODA

En nuestro ejemplo no hay nada que podar pues, como se explicó anteriormente, el diagrama FODA carece de nodos fijos.

Paso 2. Conversión de las características FODA en producciones EBNF

Aplicando la tabla de conversiones de la figura 3.5 a la figura 5.22, se obtendrá la gramática de la figura 5.34. La figura 5.35 es un ejemplo de especificación válida para esta gramática.

```
List ::= OutFile ElementType Ownership [LengthCounter] [Tracing];
OutFile ::= quotedstring;
ElementType ::= quotedstring;
Ownership ::= "External reference" | "Owned reference" | "Copy";
LengthCounter ::= LengthType;
LengthType ::= "short" | "int";
Tracing ::= "Tracing";
```

Figura 5.34. Gramática de un DSL para especificar listas.

```
"outDir/MyClassList.cpp" "MyClass" Copy short Tracing
```

Figura 5.35. Especificación de una lista según la gramática de la figura 5.34.

Paso 3. Definición, si procede, de sintaxis concretas para el DSL

Como se hizo en la sección 5.1.2.3, para simplificar el análisis de las especificaciones se desarrollará la gramática concreta de la figura 5.36, que respete la sintaxis de Ruby. La expresividad de Ruby permite crear la ilusión de escribir especificaciones abstractas, como la de la figura 5.37, cuando realmente se escribe código Ruby.

```
List ::= OutFile "," ElementType "," Ownership ["," LengthCounter] ["," Tracing];
OutFile ::= "\"Out File\"" "=>" quotedstring;
ElementType ::= "\"Element Type\"" "=>" quotedstring;
Ownership ::= "\"Ownership\"" "=>"
  ("\"External reference\"" | "\"Owned reference\"" | "\"Copy\"");
LengthCounter ::= "\"Length Counter\"" "=>" ("\"yes\"" "," LengthCounterType | "\"no\"");
LengthCounterType ::= "\"Length Counter Type\"" "=>" ("\"short\"" | "\"int\"");
Tracing ::= "\"Tracing\"" "=>" ( "\"yes\"" | "\"no\"" );
```

Figura 5.36. Gramática, equivalente a la de la figura 5.34, de un DSL interno a Ruby.

```
"Out File" => "outDir/MyClassList.cpp",
"Element Type" => "MyClass",
"Ownership" => "Copy",
"Length Counter Type" => "short",
"Tracing" => "yes"
```

Figura 5.37. Especificación de una lista según la gramática de la figura 5.36.

IV. Implementación de la flexibilización del ejemplar

Utilizando ETL se implementará una flexibilización del ejemplar que traduzca especificaciones DSL, como la figura 5.37, en listas codificadas en C++.

Las figuras 5.38 y 5.39 son el diseño y la codificación ETL de la flexibilización del ejemplar. Los generadores *ElementType, Ownership, LengthCounter* y *Tracing* implementan los requisitos variables {*ElementType*}, {*Ownership*}, {*LengthCounter, LengthType*} y {*Tracing*}, adaptando el ejemplar *List* de la figura 5.33 y produciendo, como resultado, una nueva lista *NewList*.

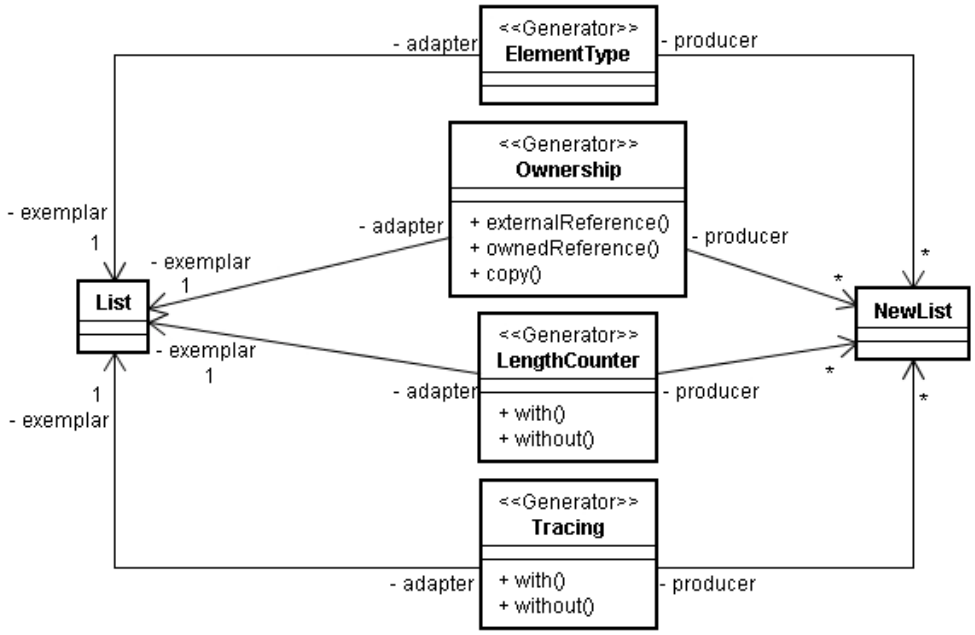


Figura 5.38. Diseño de la flexibilización ETL del ejemplar.

```
class ElementType < Generator
  def initialize(exemplar, out, elementType)
    gsub(/Exemplar/, elementType)
    prod(exemplar, out)
  end
end
```

```
class Ownership < Generator
  def initialize(exemplar, out, ownershipType)
    case ownershipType
      when 'External reference'
        externalReference
      when 'Owned reference'
        ownedReference
      when 'Copy'
        copy
    end
    prod(exemplar, out)
  end
  def externalReference
    sub(/delete head_;/, '')
    sub(/new Exemplar\(h\)/ , '&h')
  end

  def ownedReference
    sub(/new Exemplar\(h\)/ , '&h')
  end
  def copy
  end
end
```

```
class LengthCounter < Generator
  def initialize(exemplar, out, lengthType)
    if lengthType
      with(lengthType)
    else
      without
    end
    prod(exemplar, out)
  end
  def with(lengthType)
    gsub(/int/, lengthType)
  end
  def without
    gsub(/^.*length.*$/i, '')
    gsub(/\)\:/, '\0head (0), tail (t)')
  end
end
```

```
class Tracing < Generator
  def initialize(exemplar, out, tracing)
    if tracing
      with
    else
      without
    end
    prod(exemplar, out)
  end
  def with
  end
  def without
    gsub(/cout.+$/, '')
  end
end
```

ElementType	Ownership	LengthCounter	Tracing

Figura 5.39. Codificación de la flexibilización ETL del ejemplar.

Como puede apreciarse, la flexibilización ETL:

- Es no invasiva (no es necesario retocar el ejemplar).
- Es muy concisa.
- No tiene código disperso (en la figura 5.55 no hay mezcla de colores).

La figura 5.40 muestra el análisis[58] de las especificaciones DSL y la ejecución coordinada de los generadores ETL.

En la zona `MyClass` del ejemplar (figura 5.33) se solaparía la acción de los generadores *ElementType* y *Ownership*. Para evitar su colisión, estos generadores se aplicarán secuencialmente sobre el ejemplar. El resto de los generadores se podrán aplicar en paralelo

[58] La sección 5.1.2.3 explica como analizar en Ruby una especificación, escrita en un DSL interno, con el procedimiento `eval`.

con el operador +. La figura 5.41 representa mediante un diagrama de actividades el orden de aplicación expresado en la figura 5.40 que, por supuesto, no es el único válido.

```
unless ARGV[0]
  print "You should specify the list specification file"
  exit
end

# analyzer
eval "@list_specificacion = {#{File.open("#{ARGV[0]}").read}}"

# running generators
ElementType.new('List.cpp', @list_specificacion['Out File'],
  @list_specificacion['Element Type']).gen

(
  Ownership.new(@list_specificacion['Out File'], @list_specificacion['Out File'],
    @list_specificacion['Ownership'])
  +
  LengthCounter.new(@list_specificacion['Out File'], @list_specificacion['Out File'],
    @list_specificacion['Length Type'])
  +
  Tracing.new(@list_specificacion['Out File'], @list_specificacion['Out File'],
    @list_specificacion['Tracing'])
).gen
```

Figura 5.40. Análisis de las especificaciones DSL y ejecución coordinada de los generadores ETL.

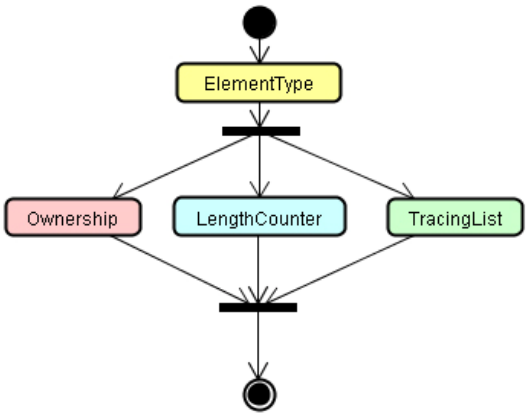


Figura 5.41. Orden de aplicación de los generadores ETL.

5.2.3. Aplicabilidad de las técnicas comunes de generalización de código a la flexibilización de un ejemplar

De la sección 5.2.3.1 a la 5.2.3.8 se mostrará el uso de diferentes técnicas internas para flexibilizar el ejemplar.

En la sección 5.1.1 se utilizó la herencia de clases para flexibilizar tipos. Con esta técnica, la parametrización se resuelve en tiempo de ejecución por enlace dinámico, lo que puede ralentizar innecesariamente los productos. En su lugar, en las secciones 5.2.3.1-5.2.3.8, se utilizará genericidad, que resuelve la parametrización en tiempo de compilación.

La sección 5.2.3.9 mostrará la flexibilización externa del ejemplar con plantillas de código.

5.2.3.1. Sentencia de selección

Quizá, el modo más inmediato de flexibilizar un ejemplar sea introducirle, de forma controlada mediante sentencias de selección, el código variable.

La figura 5.42 es una flexibilización obtenida aplicando esta técnica. Para configurar el comportamiento de una lista, se utilizará un registro de tipo *BehaviorSpecification*, definido en la figura 5.43. La figura 5.44 especifica una lista equivalente a la de la figura 5.29 instanciando un registro *BehaviorSpecification*.

```
template <class ElementType, class LengthType>
class List
{
private:
        ElementType* head_;
        List* tail_;
        LengthType length_;

        BehaviorSpecification bs_;

public:
        List(BehaviorSpecification bs, ElementType&h, List *t=0):
          head_(0), tail_(t)
        {
                bs_ = bs;
                if (bs_.lengthCounter)
                        length_ = computedLength();
                setHead(h);
        }

        ~List()
        {
                if ((bs_.ownership == OwnedReference) || (bs_.ownership == Copy))
                        delete head_;
        }

        void setHead(ElementType& h)
        {
                if (bs_.tracing)
                        cout << "setHead(" << h <<")" << endl;
                if (bs_.ownership == Copy)
                        head_ = new ElementType(h);
                else
                        head_ = &h;
        }

        ElementType& head()
        {
                if (bs_.tracing)
                        cout << "head()" << endl;
                return *head_;
        }

        void setTail(List *t)
        {
                if (bs_.tracing)
                        cout << "setTail(t)" << endl;
                tail_ = t;
                if (bs_.lengthCounter)
```

```
                        length_ = computedLength();
        }

        List *tail() const
        {
                if (bs_.tracing)
                        cout << "tail()" << endl;
                return tail_;
        }

        const LengthType& length() const
        {
                if (bs .lengthCounter)
                        return length ;
                else
                        return -1;
        }

private:
        int computedLength() const
        { return tail()?tail()->length()+1:1; }
};
```

Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.42. Flexibilización del ejemplar con sentencias de selección.

```
enum Ownership {ExternalReference, OwnedReference, Copy};
typedef struct {
        Ownership ownership;
        bool lengthCounter;
        bool tracing;
} BehaviorSpecification;
```

Figura 5.43. Registro para especificar listas según la flexibilización de la figura 5.42.

```
MyClass e;
BehaviorSpecification bs;
  bs.ownership = Copy;
  bs.lengthCounter = true;
  bs.tracing = true;
List<MyClass, short> l(bs, e);
```

Figura 5.44. Especificación de una lista, lograda parametrizando el registro de la figura 5.43.

Los principales inconvenientes de esta forma de flexibilización son:

- Mezcla los códigos fijo y variable (ver colores en la figura 5.42), lo que provoca acoplamientos *código fijo* ↔ *código variable* y *código variable* ↔ *código variable*.

- Gestiona la variabilidad en tiempo de ejecución. En nuestro ejemplo, esto provoca una ralentización innecesaria de las listas.

- Carece totalmente de modularidad.

Las próximas secciones mostrarán como obtener una flexibilización modular del ejemplar. Las secciones 5.2.3.2-5.2.3.4 utilizarán técnicas invasivas, que introducen en el ejemplar "etiquetas" que apuntan a los módulos encargados de gestionar cada requisito variable (figura 1.4). Las secciones 5.2.3.5-5.2.3.8 mostrarán el uso de técnicas no invasivas, con las que el ejemplar se mantendrá intacto y donde cada módulo se encargará, además de implementar el requisito variable correspondiente, de engancharse al ejemplar (figura 1.4).

5.2.3.2. Subprogramas

Exceptuando la variabilidad de los tipos (*ElementType* y *LengthType*) que, como se dijo anteriormente, se flexibilizará con genericidad, cada requisito variable puede implementarse con un subprograma (ver figura 5.45).

```
////////////////////////////////////////////////
// List
////////////////////////////////////////////////

template <class ElementType, class LengthType>
class List
{
private:
  ElementType* head_;
  List* tail_;
  LengthType length_;

  BehaviorSpecification bs_;
  enum Method {Constructor, Destructor, SetHead, Head, SetTail, Tail};

public:
  List(BehaviorSpecification bs, ElementType&h, List *t=0):
     head_(0), tail_(t)
  {
    bs_ = bs;
    lengthCounter(Constructor);
    setHead(h);
  }

  ~List()
  {
    ownership(*head_, Destructor);
  }

  void setHead(ElementType& h)
  {
    tracing(h, SetHead);
    ownership(h, SetHead);
  }

  ElementType& head()
  {
    tracing(*head_, Head);
    return *head_;
  }

  void setTail(List *t)
  {
    tracing(*head_, SetTail);
    tail_ = t;
    lengthCounter(SetTail);
  }

  List *tail()
  {
    tracing(*head_, Tail);
    return tail_;
  }

  const LengthType& length()
  {
    if (bs_.lengthCounter)
      return length_;
    else
      return -1;
  }

private:
```

```
  ////////////////////////////////////////////////
  // Ownership
  ////////////////////////////////////////////////
  void ownership(ElementType& h, Method caller)
  {
    switch(bs_.ownership) {
    case ExternalReference: if (caller = SetHead) head  = &h;
    case OwnedReference:
      switch(caller) {
      case Destructor: delete head ;
      case SetHead: head_ = &h;
      }
    case Copy:
      switch(caller) {
      case Destructor: delete head ;
      case SetHead: head_ = new ElementType(h);
      }
    }
  }
```

```
  ////////////////////////////////////////////////
  // LengthCounter
  ////////////////////////////////////////////////
  void lengthCounter(Method caller)
  {
    if (bs .lengthCounter)
      computedLength();
  }

  int computedLength()
  { return tail()?tail()->length()+1:1; }
```

```
  ////////////////////////////////////////////////
  // Tracing
  ////////////////////////////////////////////////
  void tracing(ElementType& h, Method caller)
  {
    if (bs .tracing)
      switch(caller) {
      case SetHead: cout << "setHead(" << h <<")" << endl;
      case Head: cout << "head()" << endl;
      case SetTail: cout << "setTail(t)" << endl;
      case Tail: cout << "tail()" << endl;
      }
  }
};
```

Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.45 Generalización del ejemplar con subprogramas.

En la figura 5.45 la implementación del requisito {*LengthCounter, LengthType*} requiere, además de genericidad para flexibilizar *LengthType*, tres subprogramas (*length, lengthCounter* y *computedLength*) y una variable global (*length_*). Cuando la implementación de un requisito variable no es trivial y requiere el uso de varios subprogramas y variables globales o estáticas, la flexibilización se vuelve poco legible y difícil de mantener. En las secciones 5.2.3.3 y 5.2.3.4 se conseguirá mayor modularidad, agrupando los subprogramas y los atributos en clases.

5.2.3.3. Composición

Las figuras 5.46 y 5.47 son el diseño y la codificación de una flexibilización donde al ejemplar (la clase *List*) se le ha quitado el código variable y se le ha introducido unos atributos *(ownership_, tracing_* y *lengthCounter_)* y unas "etiquetas" (ver código de color , , y dentro de la clase *List*) que enlazan con las clases que implementan los requisitos variables. El diseño sigue el patrón *strategy* [GHJV94], agrupando las distintas posibilidades de variación con herencia de interfaz (*Ownership* agrupa a *ExternalReference*, *OwnedReference* y *Copy*, *Tracing* agrupa a *WithTracing* y *WithoutTracing*, y *Length* agrupa a *WithLength* y *WithoutLength*). La modalidad de una lista se configura en tiempo de ejecución, pasando al constructor de la lista objetos de tipo *Ownership*, *Tracing* y *Length* para que éste actualice los atributos *ownership_, tracing_* y *lengthCounter_*. Lamentablemente, la "ejecución de las etiquetas de enlace", que consistirá en la invocación a un método de los atributos *ownership_, tracing_* o *lengthCounter_*, se resolverá con enlace dinámico, lo que ralentizará las listas. La figura 5.48 es la especificación de una lista con pertenencia de tipo *Copy*, de la que puede conocerse su longitud, y capaz de imprimir mensajes de depuración.

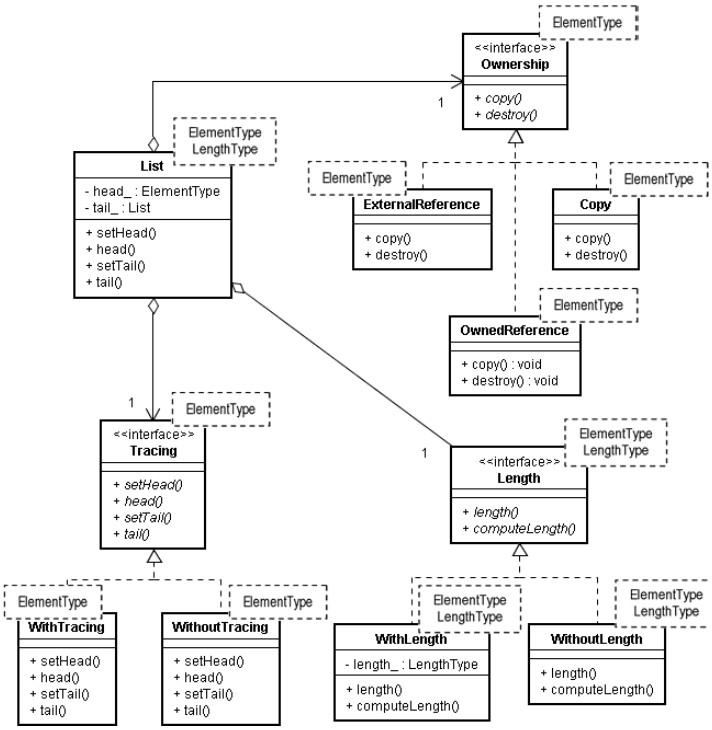


Figura 5.46. Diseño de una flexibilización del ejemplar basada en la composición y la herencia de clases.

```
//////////////////////////////////////////////////
// Ownership
//////////////////////////////////////////////////

template <class ElementType>
class Ownership {
public:
  virtual void destroy(ElementType* e) = 0;
```

```
  virtual ElementType* copy(ElementType& e) = 0;
};

template <class ElementType>
class ExternalReference : public Ownership<ElementType> {
public:
  void destroy(ElementType* e) {}
  ElementType* copy(ElementType& e) { return &e; }
};

template <class ElementType>
class OwnedReference : public Ownership<ElementType> {
public:
  void destroy(ElementType* e) { delete e;}
  ElementType* copy(ElementType& e) { return &e; }
};

template <class ElementType>
class Copy : public Ownership<ElementType> {
public:
  void destroy(ElementType* e) { delete e;}
  ElementType* copy(ElementType& e) { return new ElementType(e); }
};
```

```
////////////////////////////////////////////////
// Tracing
////////////////////////////////////////////////

template <class ElementType>
class Tracing {
public:
  virtual void setHead(ElementType& h) = 0;
  virtual void head() = 0;
  virtual void setTail() = 0;
  virtual void tail() = 0;
};

template <class ElementType>
class WithTracing : public Tracing<ElementType> {
public:
  void setHead(ElementType& h) {
    cout << "setHead(" << h << ")" << endl;
  }

  void head() {
    cout << "head()" << endl;
  }

  void setTail() {
    cout << "setTail(t)" << endl;
  }

  void tail() {
    cout << "tail()" << endl;
  }
};

template <class ElementType>
class WithoutTracing : public Tracing<ElementType> {
public:
  void setHead(ElementType& h) {}
  void head() {}
  void setTail() {}
  void tail() {}
};
```

```
////////////////////////////////////////////////
// LengthCounter interface
////////////////////////////////////////////////

template <class ElementType, class LengthType>
class LengthCounter {
public:
  virtual LengthType length() = 0;
```

```
  virtual void computeLength(void *aList) = 0;
  // due compiling limitations computeLength declares
  // (void *aList) instead of (List *aList)
  // This way, the next cyclic declaration is avoided:
  // (List declaration uses Lengthcounter declaration) &&
  // (List declaration uses Lengthcounter declaration)
};

//////////////////////////////////////////////////
// List
//////////////////////////////////////////////////

template <class ElementType, class LengthType = short>
class List
{
private:
  ElementType* head_;
  List<ElementType, LengthType>* tail ;

  Ownership<ElementType>* ownership ;
  Tracing<ElementType>* tracing ;
public:
  LengthCounter<ElementType, LengthType>* lengthCounter_;

  List(Ownership<ElementType>* ownership,
    Tracing<ElementType>* tracing,
    LengthCounter<ElementType, LengthType>* lengthCounter,
    ElementType&h, List *t=0):
    ownership_(ownership), tracing_(tracing), lengthCounter_(lengthCounter)
  {
    setHead(h);
    setTail(t);
  }

  ~List()
  {
    ownership ->destroy(head );
    delete(ownership );
    delete(tracing_);
    delete(lengthCounter_);
  }

  void setHead(ElementType& h)
  {
    tracing_->setHead(h);
    head_ = ownership_->copy(h);
  }

  ElementType& head()
  {
    tracing_->head();
    return *head_;
  }

  void setTail(List<ElementType, LengthType> *t)
  {
    tracing_->setTail();
    tail_ = t;
    lengthCounter ->computeLength(this);
  }

  List<ElementType, LengthType> *tail() const
  {
    tracing_->setTail();
    return tail ;
  }
};

//////////////////////////////////////////////////
// LengthCounter variations
//////////////////////////////////////////////////

template <class ElementType, class LengthType>
class WithLengthCounter : public LengthCounter<ElementType, LengthType> {
```

```
private:
  LengthType length ;
public:
  LengthType length()
  { return length_; }

  void computeLength(void *l)
  {
    if (static cast<List<ElementType,LengthType>*>(l)->tail()) {
      length  = static cast<List<ElementType,LengthType>*>
         (l)->tail()->lengthCounter_->length()+1;
    }
    else
      length  = 1;
  }
};

template <class ElementType, class LengthType = short>
class WithoutLengthCounter : public LengthCounter<ElementType, LengthType> {
public:
  LengthType length()
  { return -1; }

  void computeLength(void *l) {}
};
```

Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.47. Codificación de una flexibilización del ejemplar basada en la composición y la herencia de clases.

```
MyClass e;

List<MyClass, short> list(
  new Copy<MyClass>,
  new WithTracing<MyClass>,
  new WithLengthCounter<MyClass, short>,
  e
);
```

Figura 5.48. Ejemplo de especificación de una lista para la flexibilización de la figura 5.42.

5.2.3.4. Composición y genericidad

La velocidad de las listas puede incrementarse sustituyendo, en la flexibilización propuesta en la sección anterior, la herencia y la actuación del enlace dinámico por genericidad, tal como se indica en la figura 5.49. Con esta implementación, la especificación de una lista, con pertenencia de tipo *Copy*, de la que puede conocerse su longitud, y capaz de imprimir mensajes de depuración, sería como la figura 5.50.

```
//////////////////////////////////////////////////
// Ownership
//////////////////////////////////////////////////

template <class ElementType>
class ExternalReference {
public:
  void destroy(ElementType* e) {}
  ElementType* copy(ElementType& e) { return &e; }
};

template <class ElementType>
```

```
class OwnedReference {
public:
  void destroy(ElementType* e) { delete e;}
  ElementType* copy(ElementType& e) { return &e; }
};

template <class ElementType>
class Copy {
public:
  void destroy(ElementType* e) { delete e;}
  ElementType* copy(ElementType& e) { return new ElementType(e); }
};
```

```
//////////////////////////////////////////////////
// Tracing
//////////////////////////////////////////////////

template <class ElementType>
class WithTracing {
public:
  void setHead(ElementType& h) {
    cout << "setHead(" << h << ")" << endl;
  }

  void head() {
    cout << "head()" << endl;
  }

  void setTail() {
    cout << "setTail(t)" << endl;
  }

  void tail() {
    cout << "tail()" << endl;
  }
};

template <class ElementType>
class WithoutTracing {
public:
  void setHead(ElementType& h) {}
  void head() {}
  void setTail() {}
  void tail() {}
};
```

```
//////////////////////////////////////////////////
// List
//////////////////////////////////////////////////

template <class Configuration>
class List
{
private:
  Configuration::ElementType* head ;
  List<Configuration>* tail_;

  Configuration::Ownership ownership_;
  Configuration::Tracing tracing ;
public:
  Configuration::LengthCounter lengthCounter ;

  List(Configuration::ElementType&h, List<Configuration> *t=0)
  {
    setHead(h);
    setTail(t);
  }

  ~List()
  {
    ownership .destroy(head );
  }

  void setHead(Configuration::ElementType& h)
```

```
  {
    tracing .setHead(h);
    head  = ownership .copy(h);
  }

  Configuration::ElementType& head()
  {
    tracing .head();
    return *head ;
  }

  void setTail(List<Configuration> *t)
  {
    tracing .setTail();
    tail  = t;
    lengthCounter_.computeLength(this);
  }

  List *tail()
    {
    tracing .tail();
    return tail_;
  }

};
```

```
////////////////////////////////////////////////
// LengthCounter
////////////////////////////////////////////////

template <class Configuration>
class WithLengthCounter {
private:
  Configuration::LengthType length_;
public:
  Configuration::LengthType length()
  { return length ; }

  void computeLength(List<Configuration> *l)
  {
    if (static cast<List<Configuration>*>(l)->tail()) {
      length  =
        static cast<List<Configuration>*>(l)->tail()->lengthCounter .length()+1;
    }
    else
      length_ = 1;
  }
};

template <class Configuration>
class WithoutLengthCounter {
public:
  Configuration::LengthType length()
  { return -1; }

  void computeLength(List<Configuration> *l)
  {}
};
```

Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.49. Flexibilización del ejemplar basada en la composición y la herencia de clases.

```
struct Configuration
{
  typedef MyClass ElementType;
  typedef Copy<ElementType> Ownership;
  typedef WithLengthCounter<Configuration> LengthCounter;
  typedef short LengthType;
  typedef WithTracing<ElementType> Tracing;
};

MyClass e;

List<Configuration> list(e);
```

Figura 5.50. Ejemplo de especificación de una lista para la flexibilización de la figura 5.49.

5.2.3.5. Herencia múltiple

Como indica la figura 5.51, las clases que implementan los requisitos variables podrían "engancharse" de forma no invasiva al ejemplar mediante herencia. La clase padre sería el ejemplar (*List*) y sus descendientes (*ExtRefList, OwnRefList, CopyList, LengthCounterList* y *TracingList*) implementarían los requisitos variables. Un producto de la familia, por ejemplo, una lista *MyList* con pertenencia de tipo *Copy*, de la que puede conocerse su longitud y capaz de imprimir mensajes de depuración, se obtendría heredando "en paralelo" de *CopyList*, de *LengthCounterList* y de *TracingList*.

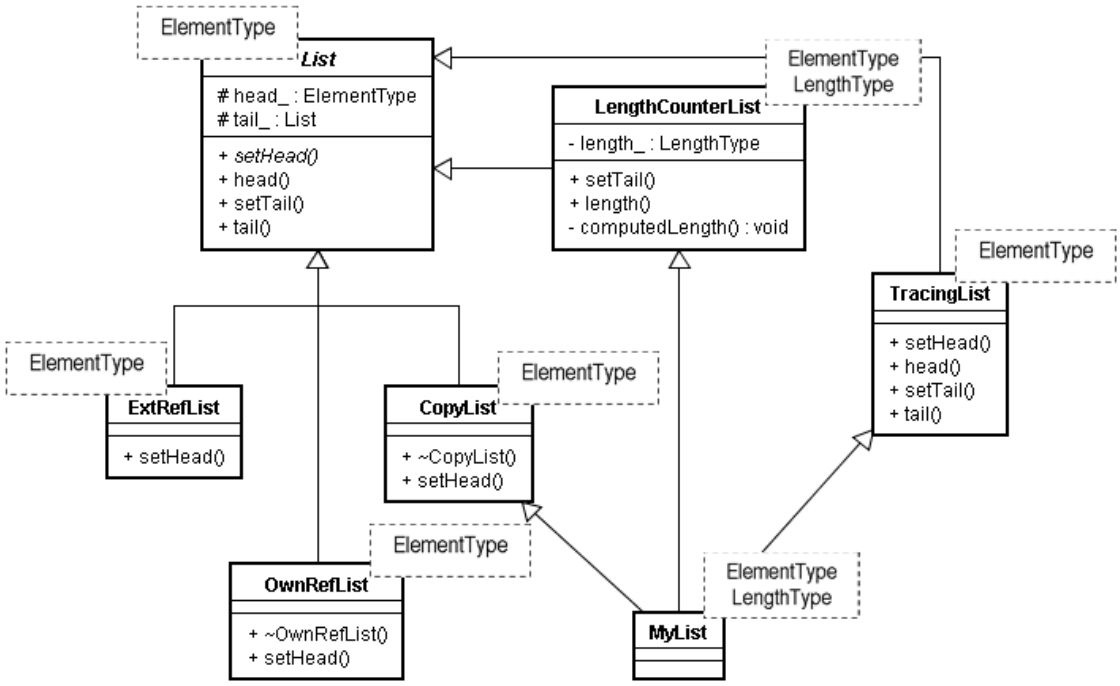


Figura 5.51. Diseño de una flexibilización del ejemplar basada en herencia múltiple.

Lamentablemente, la herencia padece las siguientes limitaciones que hacen inviable la flexibilización propuesta:

a) La herencia permite exclusivamente que las clases hijas añadan o sobreescriban métodos y atributos de las clases padres. Sin embargo tal como indica la figura 5.52, la flexibilización del ejemplar requiere:

1. Convertir en protegidos los atributos privados.

2. Eliminar atributos, expresiones, sentencias y métodos.

3. Modificar el destructor ~List, para que cuando se destruya una lista de tipo *ExternalReference* u *OwnedReference*, la invocación automática a *~List* [59], no libere los elementos de la lista.

Estas limitaciones reducen enormemente la "no invasividad" de la herencia, siendo necesaria con frecuencia la manipulación previa del ejemplar.

b) Los posibles "puntos de enganche" al ejemplar son los nombres de sus métodos y atributos. El grosor de estos puntos es excesivo e introduce acoplamientos artificiales entre el código variable que el compilador no podrá resolver. Por ejemplo, los métodos *setTail* de las clases *LengthCounterList* y *TracingList* se enganchan al método *setTail* de *List* para modificar el comportamiento de éste último. Aunque conceptualmente *LengthCounterList* y *TracingList* son independientes, y las variaciones que producen sobre *List* también lo son, un compilador será incapaz deducir cómo la clase *MyList* debe heredar el método *setTail*.

[59] *"**Automatic destructor calls**: although you are often required to make explicit constructor calls, you never need to make explicit destructor calls because there's only one destructor for any class, and it doesn't take any arguments. The compiler ensures that all destructors are called, and that means all of the destructors in the entire hierarchy, starting with the most-derived destructor and working back to the root."* [Eck00, capítulo 14]

```
class List
{
private:
        MyClass* head_;
        List* tail_;
        int length_;
public:
        List(MyClass&h, List *t=0):
          head_(0), tail_(t), length_(computedLength())
          { setHead(h); }

        ~List()
        { delete head_; }

        void setHead(MyClass& h)
        {
                cout << "setHead(" << h <<")" << endl;
                head_ = new MyClass(h);
        }

        MyClass& head()
        {
                cout << "head()" << endl;
                return *head_;
        }

        void setTail(List *t)
        {
                cout << "setTail(t)" << endl;
                tail_ = t;
                length_ = computedLength();
        }

        List *tail() const
        {
                cout << "setTail(t)" << endl;
                return tail_;
        }

        const int& length() const
        { return length_; }

private:
        int computedLength() const
        { return tail()?tail()->length()+1:1; }

};
```

Cambiar a `protected`

Eliminar si la lista no es de tipo Copy

Eliminar si la lista no tiene Tracing

Eliminar si la lista no tiene LengthCounter

Figura 5.52. Modificaciones necesarias para la flexibilización del ejemplar.

5.2.3.6. Herencia simple

Asumiendo la modificación previa del ejemplar para realizarle los cambios señalados en la figura 5.52, se podrían resolver los conflictos derivados del excesivo grosor de los puntos de enganche sustituyendo la herencia múltiple por herencia simple, tal como indica la figura 5.53.

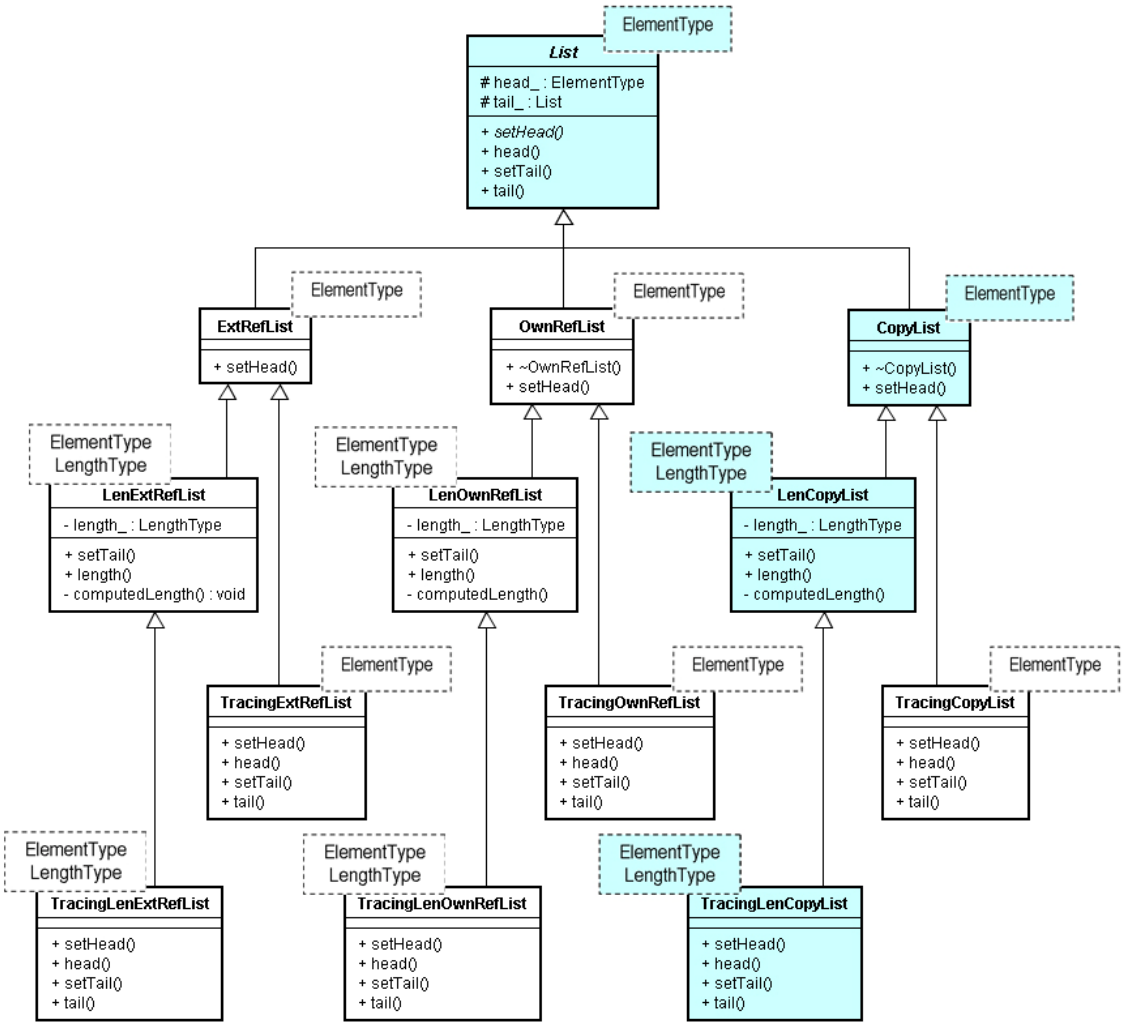


Figura 5.53. Diseño de una flexibilización del ejemplar basada en herencia simple.

Por brevedad, la figura 5.54 sólo incluye el código referente a una lista con pertenencia de tipo *Copy*, de la que puede conocerse su longitud, y capaz de imprimir mensajes de depuración (clases azules de la figura 5.53). El resto del código puede consultarse en el CDROM anexo a esta tesis.

```
template <class ElementType>
class List
{
protected:
  ElementType* head ;
  List<ElementType>* tail ;

public:
  List(ElementType&h, List<ElementType> *t=0)
  {
    setHead(h);
    setTail(t);
  }

  virtual void setHead(ElementType& h) = 0 {};

  virtual ElementType& head()
  {
```

```
    return *head ;
  }

  virtual void setTail(List<ElementType> *t)
  {
    tail_ = t;
  }

  List<ElementType> *tail() const
  { return tail ; }
};

////////////////////////////////////////////////
// Ownership
////////////////////////////////////////////////

template <class ElementType>
class CopyList : public List<ElementType>
{
public:
  CopyList(ElementType&h, CopyList<ElementType> *t=0): List<ElementType>(h, t) {}

  ~CopyList()
  { delete(head ); }

  virtual void setHead(ElementType& h)
  { head  = new ElementType(h); }
};

////////////////////////////////////////////////
// LengthCounter
////////////////////////////////////////////////

template <class ElementType, class LengthType>
class LenCopyList : public CopyList<ElementType>
{
public:
  LenCopyList(ElementType&h, LenCopyList<ElementType, LengthType> *t=0):
    CopyList<ElementType>(h, t)
  {
    length  = computedLength();
  }

  virtual void setTail(LenCopyList<ElementType, LengthType> *t)
  {
    CopyList<ElementType>::setTail(t);
    length  = computedLength();
  }

  LengthType length()
  { return length_; }

private:
  LengthType length ;
  LengthType computedLength()
  {
    if (tail())
      return static_cast<LenCopyList<ElementType, LengthType>*>(tail())->length()+1;
    else
      return 1;
  }
};

////////////////////////////////////////////////
// Tracing
////////////////////////////////////////////////

template <class ElementType, class LengthType>
class TracingLenCopyList : public LenCopyList<ElementType, LengthType>
{
public:
  TracingLenCopyList(ElementType&h, TracingLenCopyList<ElementType, LengthType> *t=0):
    LenCopyList<ElementType, LengthType>(h, t) {}
```

```
  void setHead(ElementType& h)
  {
    cout << "setHead(" << h << ")" << endl;
    LenCopyList<ElementType, LengthType>::setHead(h);
  }

  ElementType& head()
  {
    cout << "head()" << endl;
    return LenCopyList<ElementType, LengthType>::head();
  }

  void setTail(TracingLenCopyList<ElementType, LengthType> *t)
  {
    cout << "setTail(t)" << endl;
    LenCopyList<ElementType, LengthType>::setTail(t);
  }

  TracingLenCopyList<ElementType, LengthType> *tail() const
  {
    cout << "tail()" << endl;
    return static_cast<TracingLenCopyList<ElementType, LengthType>*>
      (LenCopyList<ElementType, LengthType>::tail());
  }
};
```

Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.54. Extracto de la codificación de una generalización del ejemplar basada en herencia simple.

5.2.3.7. Herencia parametrizada

Para evitar la descripción exhaustiva y redundante de cada posible combinación de requisitos variables, puede logarse un híbrido entre herencia simple y herencia múltiple enriqueciendo la herencia simple con genericidad. Concretamente, las plantillas de C++ permiten parametrizar el padre de una clase. Las figuras 5.55 y 5.56 muestran, respectivamente, el diseño detallado y la codificación del espacio del problema mediante herencia parametrizada.

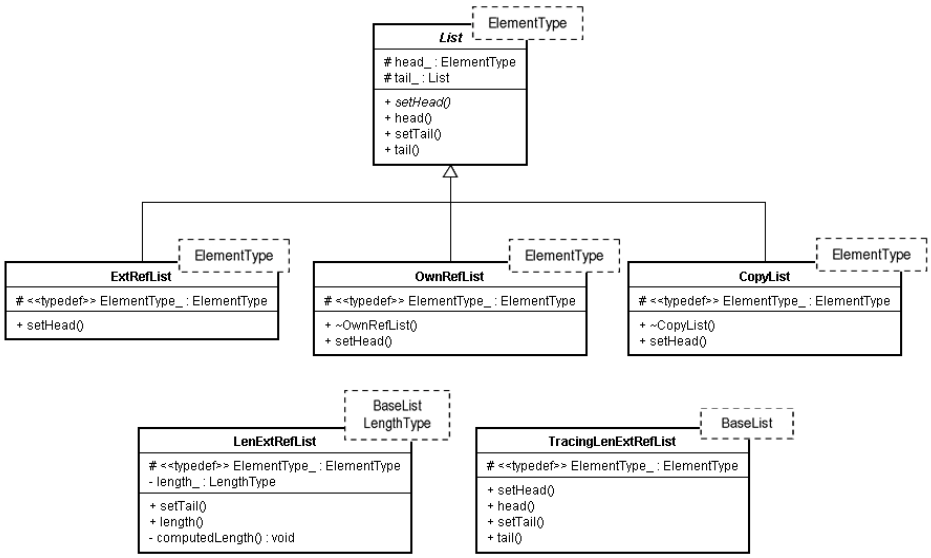


Figura 5.55. Diseño de una flexibilización del ejemplar basada en herencia parametrizada.

```
////////////////////////////////////////////////
// List
////////////////////////////////////////////////

template <class ElementType>
class List
{
protected:
  ElementType* head ;
  List<ElementType>* tail ;

public:
  List(ElementType&h, List<ElementType> *t=0)
  {
    setHead(h);
    setTail(t);
  }

  virtual void setHead(ElementType& h) = 0 {};

  virtual ElementType& head()
  {
    return *head_;
  }

  virtual void setTail(List<ElementType> *t)
  {
    tail_ = t;
  }

  List<ElementType> *tail() const
  { return tail ; }
};
```

```
////////////////////////////////////////////////
// Ownership
////////////////////////////////////////////////

template <class ElementType>
class ExtRefList : public List<ElementType>
{
protected:
  typedef ElementType ElementType ;

public:
  ExtRefList(ElementType&h, ExtRefList<ElementType> *t=0): List<ElementType>(h, t) {}

  virtual void setHead(ElementType& h)
  { head  = &h; }
};

template <class ElementType>
class OwnRefList : public List<ElementType>
{
protected:
  typedef ElementType ElementType ;

public:
  OwnRefList(ElementType&h, OwnRefList<ElementType> *t=0): List<ElementType>(h, t) {}

  ~OwnRefList()
  { delete(head ); }

  virtual void setHead(ElementType& h)
  { head  = &h; }
};

template <class ElementType>
class CopyList : public List<ElementType>
{
protected:
  typedef ElementType ElementType_;
```

```
public:
  CopyList(ElementType&h, CopyList<ElementType> *t=0): List<ElementType>(h, t) {}

  ~CopyList()
  { delete(head_); }

  virtual void setHead(ElementType& h)
  { head  = new ElementType(h); }
};
```

```
////////////////////////////////////////////////
// LengthCounter
////////////////////////////////////////////////

template <class BaseList, class LengthType>
class LengthList : public BaseList
{
protected:
  typedef BaseList::ElementType  ElementType ;

public:
  LengthList(ElementType_&h, LengthList<BaseList, LengthType> *t=0):
    BaseList(h, t)
  {
    length  = computedLength();
  }

  virtual void setTail(LengthList<BaseList, LengthType> *t)
  {
    BaseList::setTail(t);
    length  = computedLength();
  }

  LengthType length()
  { return length ; }

private:
  LengthType length_;
  LengthType computedLength()
  {
    if (tail())
      return static cast<LengthList<BaseList, LengthType>*>(tail())->length()+1;
    else
      return 1;
  }
};
```

```
////////////////////////////////////////////////
// Tracing
////////////////////////////////////////////////

template <class BaseList>
class TracingList : public BaseList
{
protected:
  typedef BaseList::ElementType_ ElementType_;

public:
  TracingList(ElementType &h, TracingList<BaseList> *t=0): BaseList(h, t) {}

  void setHead(ElementType_& h)
  {
    cout << "setHead(" << h << ")" << endl;
    BaseList::setHead(h);
  }

  ElementType_& head()
  {
    cout << "head()" << endl;
    return BaseList::head();
  }

  void setTail(TracingList<BaseList> *t)
```

```
  {
    cout << "setTail(t)" << endl;
    BaseList::setTail(t);
  }

  TracingList<BaseList> *tail() const
  {
    cout << "tail()" << endl;
    return static cast<TracingList<BaseList>*>(BaseList::tail());
  }

};
```

Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.56. Codificación de una generalización del ejemplar basada en herencia parametrizada.

Para obtener, por ejemplo, una lista con pertenencia de tipo *Copy*, de la que puede conocerse su longitud y capaz de imprimir mensajes de depuración, bastaría con escribir la siguiente sentencia:

```
typedef TracingList< LengthList<CopyList<MyClass>, short> > MyList;
```

5.2.3.8. Orientación a aspectos

AspectC++ [SLU05] es una extensión orientada a aspectos de C++ análoga a AspectJ, el lenguaje de orientación a aspectos más difundido en la actualidad (ver sección 2.1.3.1). Utilizando AspectC++ podría considerarse que un ejemplar es el concepto central de una familia de productos y que los requisitos variables son aspectos que se superponen a él.

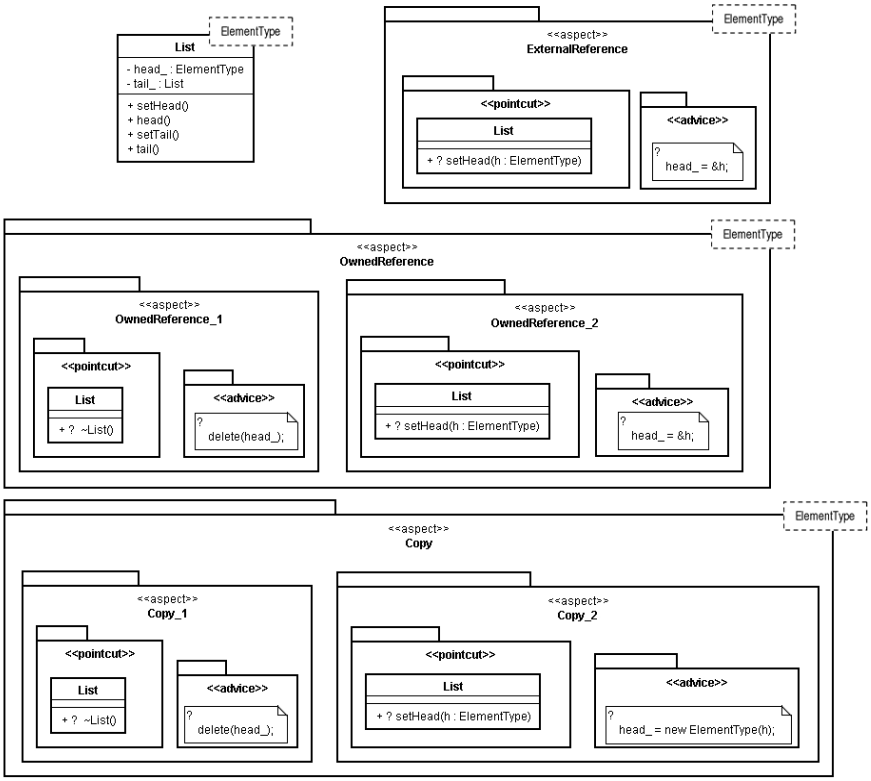
En este caso, los puntos de enganche serían los *join points* que pueden seleccionarse con un *pointcut*. Afortunadamente, el grosor de los *join points* es suficientemente fino (llamadas a métodos, accesos a atributos, inicialización de objetos…) como para evitar los acoplamientos entre el código variable que introduce la herencia.

Sin embargo, las extensiones de orientación a aspectos para GPLs suelen sufrir limitaciones que impiden flexibilizaciones realmente no invasivas del ejemplar. Concretamente, AspectC++ sólo permite los siguientes cambios estructurales (denominados *static crosscutting)*: la adición de atributos y métodos a una clase, y la alteración del padre de una clase. Es decir, los mismos cambios que la herencia parametrizada. Por lo tanto, las modificaciones señaladas en la figura 5.52 no podrían realizarse[60].

Si se retocara el ejemplar para quitarle el código variable, se podría plantear la flexibilización de la figura 5.57. En esta figura, se emplea el perfil UML propuesto en

[60] Si el ejemplar estuviera escrito en Java, con AspectJ tampoco podríamos especificar en un aspecto los cambios citados [AJPG06].

[Zha05]. Los aspectos son paquetes con el estereotipo `<<aspect>>`, que pueden contener otros aspectos o un *pointcut* y su *advice* asociado, los cuales se representan como paquetes con los estereotipos `<<pointcut>>` y `<<advice>>`. La clase *List* sería el ejemplar retocado, que implementaría exclusivamente los requisitos fijos. Añadiéndole los aspectos `[ExternalReference|OwnedReference|Copy][LengthCounter][Tracing]` se modificaría su comportamiento para adaptarlo a la especificación de cualquier producto. Lamentablemente, esta flexibilización seguiría siendo irrealizable porque en AspectC++ los aspectos no pueden ser genéricos ni actuar sobre clases genéricas[61, 62] y la implementación de los tipos variables *ElementType* y *LengthCounterType* requeriría que los aspectos *ExternalReference*, *OwnedReference*, *Copy*, *LengthCounter* y *Tracing* fueran plantillas y que, además, los aspectos actuaran sobre otra plantilla, la clase *List*.



[61] *"Currently ac++ is able to parse a lot of the (really highly complicated) C++ templates, but weaving is restricted to non-templated code only. That means you can not weave in templates or even affect calls to template functions or members of template classes."* [Spi06, página 19]

[62] Aunque la versión 1.5.2 de AspectJ permite el desarrollo de aspectos genéricos y la adición de aspectos sobre clases genéricas, la propia genericidad de Java posee importantes limitaciones: *"All instances of a generic class have the same run-time class, regardless of their actual type parameters […] As consequence, the static variables and methods of a class are also shared among all the instances. That is why it is illegal to refer to the type parameters of a type declaration in a static method or initializer, or in the declaration or initializer of a static variable."* [Bra04, página 14]

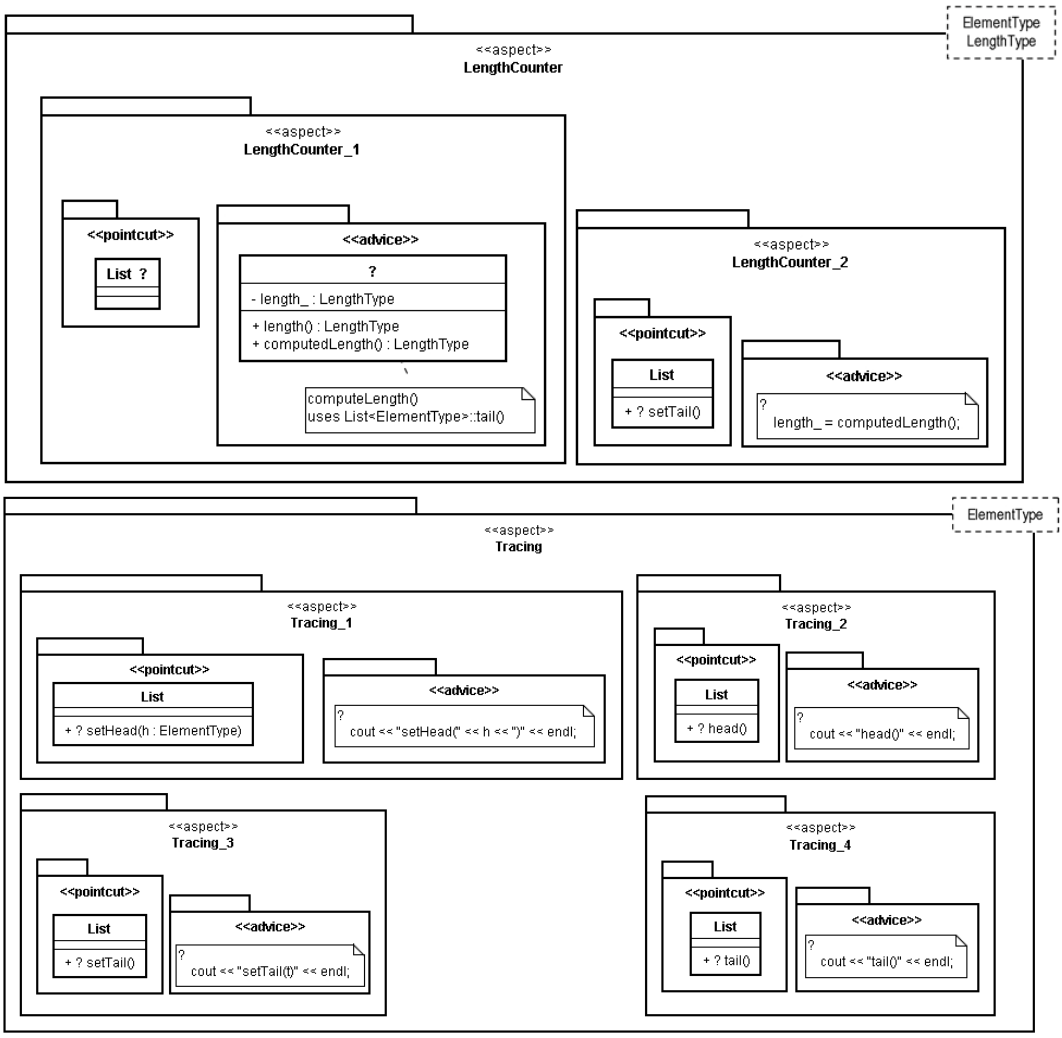


Figura 5.57. Diseño, según la notación propuesta en [Zha05], de una flexibilización del ejemplar basada en orientación a aspectos.

5.2.3.9. Plantillas de código

Las plantillas de código gozan de gran difusión y suelen emplearse para generar código ejecutable (herramientas MDA como *AndroMDA* [AM06], *CodaGen Architect* [CA06]…), código HTML (tecnologías JSP, ASP…), etc.

La figura 5.58 es la codificación de una flexibilización externa, invasiva y no modular del ejemplar, realizada con la tecnología de plantillas ERB [ERB06]. Con esta tecnología, el código variable (colores □, □, □ y □) se escribe en Ruby y se incrusta, delimitado por los símbolos <% y %>, en el código fijo (color blanco).

```
class <%=@list_specificacion['Element Type']%>List
{
private:
  <%=@list_specificacion['Element Type']%>* head_;
  <%=@list_specificacion['Element Type']%>List* tail_;
  <% if @list_specificacion['Length Counter Type']%>
  <%=@list_specificacion['Length Counter Type']%> length_;
  <% end %>
public:
  <%=@list_specificacion['Element Type']%>List
    (<%=@list_specificacion['Element Type']%>&h,
     <%=@list_specificacion['Element Type']%>List *t=0):
    head_(0), tail_(t)
    <% if @list_specificacion['Length Counter Type']%>
    , length_(computedLength())
    <% end %>
    { setHead(h); }

  ~<%=@list_specificacion['Element Type']%>List()
  {
  <% if @list_specificacion['Ownership'] == 'Owned reference' or
    @list_specificacion['Ownership'] == 'Copy'%>
  delete head_;
  <% end %>
  }

  void setHead(<%=@list_specificacion['Element Type']%>& h)
  {
    <% if @list_specificacion['Tracing'] %>
    cout << "setHead(" << h <<")" << endl;
    <% end %>
    head_ =
      <% if @list_specificacion['Ownership'] == 'External reference' or
        @list_specificacion['Ownership'] == 'Owned reference'%>
      &h;
      <% else %>
      new <%=@list_specificacion['Element Type']%>(h);
      <% end %>
  }

  <%=@list_specificacion['Element Type']%>& head()
  {
    <% if @list_specificacion['Tracing'] %>
    cout << "head()" << endl;
    <% end %>
    return *head_;
  }

  void setTail(<%=@list_specificacion['Element Type']%>List *t)
  {
    <% if @list_specificacion['Tracing'] %>
    cout << "setTail(t)" << endl;
    <% end %>
    tail_ = t;
    <% if @list_specificacion['Length Counter Type']%>
    length_ = computedLength();
    <% end %>
  }

  <%=@list_specificacion['Element Type']%>List *tail() const
  {
        <% if @list_specificacion['Tracing'] %>
                cout << "tail()" << endl;
        <% end %>
        return tail_;
  }

  <% if @list_specificacion['Length Counter Type']%>
  const <%=@list_specificacion['Length Counter Type']%>& length() const
  { return length_; }
```

```
  <%=@list_specificacion['Length Counter Type']%> computedLength() const
  { return tail()?tail()->length()+1:1; }
  <% end %>
};
```

Código	Metacódigo			
Funcionalidad fija de todas las listas	ElementType	Ownership	LengthCounter	Tracing

Figura 5.58. Flexibilización del ejemplar con plantillas ERB.

La figura 5.59 muestra el análisis de las especificaciones DSL[63] y la generación de los productos correspondientes, invocando al motor ERB con la plantilla de la figura 5.58.

```
require "erb/erb"

unless ARGV[0]
        print "You should specify the list specification file"
        exit
end

#####################################
# analyzer
#####################################

eval "@list_specificacion = {#{File.open("#{ARGV[0]}").read}}"

#####################################
# generator call
#####################################

erb = ERB.new(File.open("TemplateList.cpp").read)
list_code = erb.result(binding)
File.open(@list_specificacion['Out File'], "w").write(list_code)
```

Figura 5.59. Análisis de especificaciones DSL y generación de listas con plantillas ERB.

Las plantillas de código son la técnica externa análoga a las sentencias de selección (sección 5.2.3.1). Como aquel tipo de flexibilización, produce líneas de productos monolíticas y con excesivo acoplamiento. Estos inconvenientes pueden atenuarse extrayendo de la plantilla el código variable y modularizándolo con subprogramas y clases, de manera similar a como se planteó en las secciones 5.2.3.2-5.2.3.4.

5.3. Conclusiones

En la sección 5.1 se ha planteado un ejemplo que resalta dos factores clave de las familias de productos: el nivel de ocultación de la infraestructura que facilita la obtención automática de los productos y la eficiencia de los productos finales.

[63] La gramática del DSL se describió en la figura 5.36.

Se ha visto que la ocultación permite abstraer la especificación de los productos de la implementación de la infraestructura. Esto facilita el aprendizaje y el manejo de la infraestructura, así como la evolución independiente de las especificaciones y de la implementación de las familias de productos.

Normalmente, el valor de los requisitos variables de un producto puede concretarse antes de su ejecución. La sección 5.1.2 ha mostrado cómo aprovechar este hecho, haciendo que las infraestructuras sean compiladores en vez de intérpretes.

En la sección 5.2 se ha abordado la resolución de un ejemplo aplicando dos metodologías distintas: la propuesta en [CE00] y EDD.

La tendencia metodológica general para el desarrollo de familias de productos consiste en plantear una ampliación de la ingeniería del software clásica, donde las fases del ciclo de vida se adapten para cubrir la creación de familias de productos en lugar de productos específicos. En la sección 5.2.1 se ha resumido la aplicación de la metodología propuesta en [CE00] a un ejemplo. Con esta metodología, el análisis aprovecha técnicas novedosas, como los diagramas FODA. Sin embargo, las fases de diseño y codificación no incorporan aportaciones significativas respecto a las técnicas empleadas tradicionalmente en ingeniería del software para el desarrollo de productos aislados.

La metodología EDD, original de esta tesis, propone una estrategia innovadora que trata de sacar partido a la similitud entre los productos de una familia afrontando su construcción por analogía respecto a un ejemplar de la familia realizado anteriormente. Así, el desarrollo se descompone en dos grandes etapas: la construcción de un ejemplar y su posterior flexibilización. Mientras que la primera etapa aprovecha la madurez de la ingeniería del software clásica para implementar los requisitos fijos del dominio, presentes en cualquier producto, la segunda busca cómo derivar automáticamente del ejemplar los demás productos de la familia.

En las secciones 5.2.3.1-5.2.3.9 se ha examinado cómo flexibilizar un ejemplar mediante las técnicas que tradicionalmente se utilizan para generalizar código. Lamentablemente, se ha visto que estas técnicas padecen serias limitaciones que justifican la necesidad de ETL.

En las secciones 5.1.2.3 y 5.2.2 se han implementado con ETL dos flexibilizaciones satisfactorias, que gozan de las siguientes cualidades:

- Son realmente no invasivas.

- No introducen acoplamientos "artificiales" entre el código variable ni del código fijo al código variable.

- Son muy concisas.

- Ofrecen una dispersión mínima del código variable. Compárese la distribución de los colores □, □, □ y □ en la figura 5.39 con la distribución en las figuras 5.42, 5.45, 5.47, 5.49, 5.54, 5.56, 5.58 o en la figura 5.28 (la familia de productos que se construyó según la metodología de [CE00]).

-

6

Ejemplos de aplicación

I would rather write programs to help me write programs than write programs.

R. L. Sites, *Proving that computer programs terminate cleanly.*

El presente capítulo incluye cinco ejemplos que ilustran la potencia y versatilidad de EDD y ETL.

El primer ejemplo consiste en el desarrollo de una herramienta de generación automática de documentación HTML para código SQL, similiar a *Javadoc* [Jav06].

El segundo ejemplo resuelve el desarrollo automático de lectores Java de ficheros CSV. Además de generar código ejecutable, también se produce documentación y juegos de prueba.

El tercer ejemplo aborda la generación automática de procedimientos almacenados en TRANSACT SQL. En este ejemplo, la parametrización de la generación se automatiza mediante reflexión, es decir, consultando la metainformación del gestor de base de datos SQL Server.

El cuarto ejemplo consiste en la generación automática de pruebas de unidades para programas escritos en Modula-2.

Por último, el quinto ejemplo muestra cómo extender la sintaxis de Java para que este lenguaje sea más conciso.

Los ejemplos primero, segundo y quinto se inspiran en supuestos prácticos planteados por J. Herrington en [Her03]. El doctor Eugenio Arellano Alameda suministró el enunciado del tercer ejemplo, que procede de su trabajo en Informática Presupuestaria. El cuarto ejemplo es completamente original.

6.1. Documentación de código

Uno de los objetivos de cualquier lenguaje de programación es promover la escritura de código legible. Lamentablemente, con los lenguajes actuales sigue siendo difícil escribir código que se autodocumente y muestre la intención del programador [CE00, capítulo 11; IS06]. Por esta razón, es común acompañar el código con documentación que lo complemente.

La documentación del código mediante comentarios inscritos en él presenta algunos inconvenientes:

- La consulta de la documentación se vuelve tediosa porque está mezclada con el código.

- Para disponer de la documentación es imprescindible contar con el código que la contiene.

Como alternativa, el código puede documentarse por separado. Sin embargo, como señalan los defensores de los procesos ágiles, la documentación "en papel" suele ser costosa y propensa a contener incongruencias con el código, ya que la evolución del código no siempre se refleja en el papel [Bec02, página 159].

Una solución de compromiso es incluir la documentación en el código y utilizar después una herramienta que, de forma automática, extraiga la documentación y la formatee adecuadamente. Así,

- la cercanía física de la documentación al código agiliza la propagación de los cambios del código a la documentación.

- la extracción y el formateo automático ofrecen una documentación legible y manejable a un coste mínimo.

Desde que Java adoptara esta solución, proporcionando la herramienta *Javadoc* [Jav06] muchos lenguajes han incorporado herramientas análogas, como *Rdoc* para Ruby [Rdo07], *POD* para PERL [POD06]…

En esta sección se aborda la construcción con EDD y ETL de una herramienta de este tipo, llamada SQLDoc, que facilitará la documentación de código SQL.

6.1.1. Análisis de la familia de productos

Para mejorar la comprensión del ejemplo, nos limitaremos a automatizar la extracción y el formateo del código SQL relativo a la creación de tablas. SQLDoc extraerá la documentación embebida y la presentará en formato **HTML**. La definición de cada tabla llevará asociada comentarios que describirán:

- la tabla.
- cada campo o columna de la tabla.
- las relaciones de la tabla con otras tablas (a través de claves ajenas).

La figura 6.1 es un diagrama FODA que modela la variabilidad que asumirá SQLDoc.

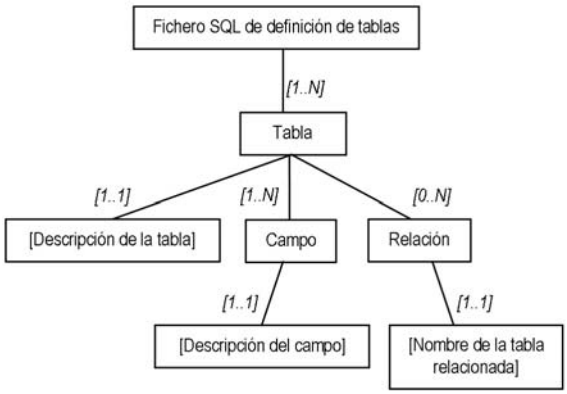


Figura 6.1. Model del dominio "documentación de la creación de tablas en SQL".

6.1.2. Construcción de un ejemplar

Para la creación de SQLDoc se desarrollará, siguiendo los criterios enunciados en la sección 3.2.1.6, un ejemplar del tipo de documentación HTML que se desea generar. La figura 6.2 muestra un posible ejemplar, incluido en el CDROM anexo a esta tesis, que documentaría una base de datos sobre libros. Concretamente, las figuras 6.3, 6.4 y 6.5 son los ficheros HTML sobre los que se apoyará SQLDoc para la generación.

Book
Author
Publisher

Table Name: Book

Description: Book contains information about each book, the title, the author, the publisher, etc.

Fields:

Name	Type	Constraints	Comments
bookID	integer	Non-null	The primary key id
title	varchar(80)	Non-null	The title of the book
ISBN	varchar(80)	Non-null, Unique	The ISBN number of the book
authorID	integer	Non-null	The author (as a foreign key relation)
publisherID	integer	Non-null	The publisher (as a foreign key relation)
status	integer	Non-null	The active or inactive state of the record
numCopies	integer	Non-null	

Relates To: Author, Publisher

Figura 6.2. Representación en un navegador del ejemplar de SQLDoc.

```
<frameset cols="20%,*">
        <frame src="tables.html">
        <frame name="main" src="tables.html">
</frameset>
```

Figura 6.3. Ejemplar de SQLDoc (index.html).

```
<html>
<body>

<a href="Book.html" target="main">Book</a><br>

<a href="Author.html" target="main">Author</a><br>

<a href="Publisher.html" target="main">Publisher</a><br>

</body>
</html>
```

Figura 6.4.Ejemplar de SQLDoc (Tables.html).

```
<html><head>
<title>Book</title>
</head>
<body>
<table width=100%>
<tr>
<td width=10% nowrap valign=top>6.2Table Name:</td><td>Book</td></tr>
<td width=10% nowrap valign=top>6.2Description:</td><td>Book contains information about each
book, the title, the author, the publisher, etc.</td></tr>
<td width=10% nowrap valign=top>6.2Fields:</td><td>
<table cellspacing=1 cellpadding=0 bgcolor=#bbbbbb width=100%>
<tr>
<td width=20% nowrap valign=top>6.2Name</b></td>
<td width=20% nowrap valign=top>6.2Type</b></td>
<td width=20% nowrap valign=top>6.2Constraints</b></td>
<td width=40% valign=top>6.2Comments</b></td>
</tr>
<tr>
<td bgcolor=white valign=top>bookID</td>
<td bgcolor=white valign=top>integer</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top>The primary key id</td>
</tr><tr>
<td bgcolor=white valign=top>title</td>
<td bgcolor=white valign=top>varchar(80)</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top>The title of the book</td>
</tr><tr>
```

```
<td bgcolor=white valign=top>ISBN</td>
<td bgcolor=white valign=top>varchar(80)</td>
<td bgcolor=white valign=top>Non-null, Unique</td>
<td bgcolor=white valign=top>The ISBN number of the book</td>
</tr><tr>
<td bgcolor=white valign=top>authorID</td>
<td bgcolor=white valign=top>integer</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top>The author (as a foreign key relation)</td>
</tr><tr>
<td bgcolor=white valign=top>publisherID</td>
<td bgcolor=white valign=top>integer</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top>The publisher (as a foreign key relation)</td>
</tr><tr>
<td bgcolor=white valign=top>status</td>
<td bgcolor=white valign=top>integer</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top>The active or inactive state of the record</td>
</tr><tr>
<td bgcolor=white valign=top>numCopies</td>
<td bgcolor=white valign=top>integer</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top></td>
</tr>
</table>
</td></tr>
<td width=10% nowrap valign=top>6.2Relates To:</td>
<td valign=top><a href="Author.html">Author</a>, <a href="Publisher.html">Publisher</a></td>
</tr></table>
</body>
</html>
```

Figura 6.5. Ejemplar de SQLDoc (Book.html).

6.1.3. Definición de una interfaz para la especificación abstracta de los productos

Aplicando el proceso descrito en la sección 3.2.2.1.1 a la figura 6.1, se obtendría la sintaxis abstracta de la figura 6.6 para un DSL con que documentar la creación de tablas. Puesto que las instrucciones del DSL irán embebidas como comentarios en el código SQL, se utilizará la sintaxis concreta de la figura 6.7.

```
FicheroSQL ::= {Tabla};
Tabla ::= DescripcionTabla {Campo} [{Relacion}];
DescripcionTabla ::= '[a-zA-Z_ ()]+';
Campo ::= DescripcionCampo;
Relacion ::= NombreTablaRelacionada;
DescripcionCampo ::= '[a-zA-Z_ ()]+';
NombreTablaRelacionada ::= '[a-zA-Z_]+';
```

Figura 6.6. Sintaxis abstracta de un DSL para documentar la creación de tablas en SQL.

```
FicheroSQL ::= {Tabla};
Tabla ::= DescripcionTabla {Campo} [{Relacion}];
DescripcionTabla ::= '-- @desc ' '[a-zA-Z0-9_ ():,.]+';
Campo ::= DescripcionCampo;
Relacion ::= NombreTablaRelacionada;
```

```
DescripcionCampo ::= '-- @field ' '[a-zA-Z0-9_ ():,.]+';
NombreTablaRelacionada ::= '-- @relates_to ' '[a-zA-Z0-9_ ():,.]+';
```

Figura 6.7. Sintaxis concreta de un DSL para documentar la creación de tablas en SQL.

La figura 6.8 es un ejemplo de fichero SQL con documentación embebida. Las figuras 6.9 y 6.10 son parte[64] de la documentación que generaría SQLDoc a partir de dicho ejemplo.

```
-- @desc Clientes de un banco
-- @field nombre Nombre de un cliente (clave primaria)
-- @field calle Domicilio donde está empadronado el cliente
-- @field localidad Localidad donde está empadronado el cliente
create table Cliente (
  nombre varchar(80) not null primary key
  ,calle varchar(40) not null
  ,localidad varchar(20) not null
);

-- @desc Sucursales de un banco
-- @field nombre Nombre de una sucursal (clave primaria)
-- @field localidad Localidad donde se ubica la sucursal
-- @field activos Dinero que la sucursal dispone a diario en efectivo
create table Sucursal (
  nombre varchar(80) not null primary key
  ,localidad varchar(40) not null
  ,activos integer not null
);

-- @desc Cuentas bancarias
-- @field numero Identificador de una cuenta (clave primaria)
-- @field nombre Nombre de la sucursal a la que está asociada la cuenta
-- @field saldo Saldo disponible en la cuenta
-- @relates_to Sucursal
create table Cuenta (
  numero integer not null primary key
  ,nombre varchar(80) not null
  ,saldo integer not null
);

-- @desc Relación N:M entre los clientes de un banco y sus cuentas
-- @field nombre Nombre de un cliente (clave primaria)
-- @field numero Identificador de una cuenta (clave primaria)
-- @relates_to Cliente
-- @relates_to Cuenta
create table Impositor (
  nombre varchar(80) not null
  ,numero integer not null
);

alter table Impositor
    add constraint Impositor_claves_primarias
    primary key (nombre, numero);

alter table Cuenta
    add constraint Cuenta_Sucursal
    foreign key (nombre)
    references Sucursal (nombre);

alter table Impositor
```

[64] Como se aprecia en la figura 6.11, SQLDoc generará para el ejemplo de la figura 6.8 los ficheros *index.html, tables.html, Cliente.html, Impositor.html, Cuenta.html* y *Sucursal.html.*

```
    add constraint Impositor_Cliente
    foreign key (nombre)
    references Cliente (nombre);

alter table Impositor
    add constraint Impositor_Cuenta
    foreign key (numero)
    references Cuenta (numero);
```

Figura 6.8. Ejemplo de código SQL documentado (banco.sql)

```
<html>
<body>

<a href="Cliente.html" target="main">Cliente</a><br>

<a href="Sucursal.html" target="main">Sucursal</a><br>

<a href="Cuenta.html" target="main">Cuenta</a><br>

<a href="Impositor.html" target="main">Impositor</a><br>

</body>
</html>
```

Figura 6.9. Generación obtenida por SQLDoc a partir de la figura 6.8 (Tables.html).

```
<html><head>
<title>Cuenta</title>
</head>
<body>
<table width=100%>
<tr>
<td width=10% nowrap valign=top>6.2Table Name:</td><td>Cuenta</td></tr>
<td width=10% nowrap valign=top>6.2Description:</td><td>Cuentas bancarias</td></tr>
<td width=10% nowrap valign=top>6.2Fields:</td><td>
<table cellspacing=1 cellpadding=0 bgcolor=#bbbbbb width=100%>
<tr>
<td width=20% nowrap valign=top>6.2Name</b></td>
<td width=20% nowrap valign=top>6.2Type</b></td>
<td width=20% nowrap valign=top>6.2Constraints</b></td>
<td width=40% valign=top>6.2Comments</b></td>
</tr>
<tr>
<td bgcolor=white valign=top>numero</td>
<td bgcolor=white valign=top>integer</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top>Identificador de una cuenta (clave primaria)</td>
</tr><tr>
<td bgcolor=white valign=top>nombre</td>
<td bgcolor=white valign=top>varchar(80)</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top>Nombre de la sucursal a la que está asociada la cuenta</td>
</tr><tr>
<td bgcolor=white valign=top>saldo</td>
<td bgcolor=white valign=top>integer</td>
<td bgcolor=white valign=top>Non-null</td>
<td bgcolor=white valign=top>Saldo disponible en la cuenta</td>
</tr></table>
</td></tr>
<td width=10% nowrap valign=top>6.2Relates To:</td>
<td valign=top><a href="Sucursal.html">Sucursal</a></td>
</tr></table>
</body>
</html>
```

Figura 6.10. Generación obtenida por SQLDoc a partir de la figura 6.8 (Cuenta.html).

6.1.4. Implementación de la flexibilización del ejemplar

La figura 6.11 resume la arquitectura de SQLDoc y muestra cómo actuaría para obtener la documentación HTML correspondiente al ejemplo de la figura 6.8.

El **analizador** puede consultarse en el CD anexo a esta tesis y aprovecha las librerías proporcionadas por J. Herrington en [Her03].

Finalmente, las figuras 6.12 y 6.13 son respectivamente el diseño y la codificación del **generador**. Se han utilizado los colores ○, ○, ○, ○, ○ y ○ para identificar el código variable en el ejemplar (figuras 6.4 y 6.5), las sustituciones que lo adaptarán (figura 6.13) y el código objeto que producirán (figuras 6.9 y 6.10).

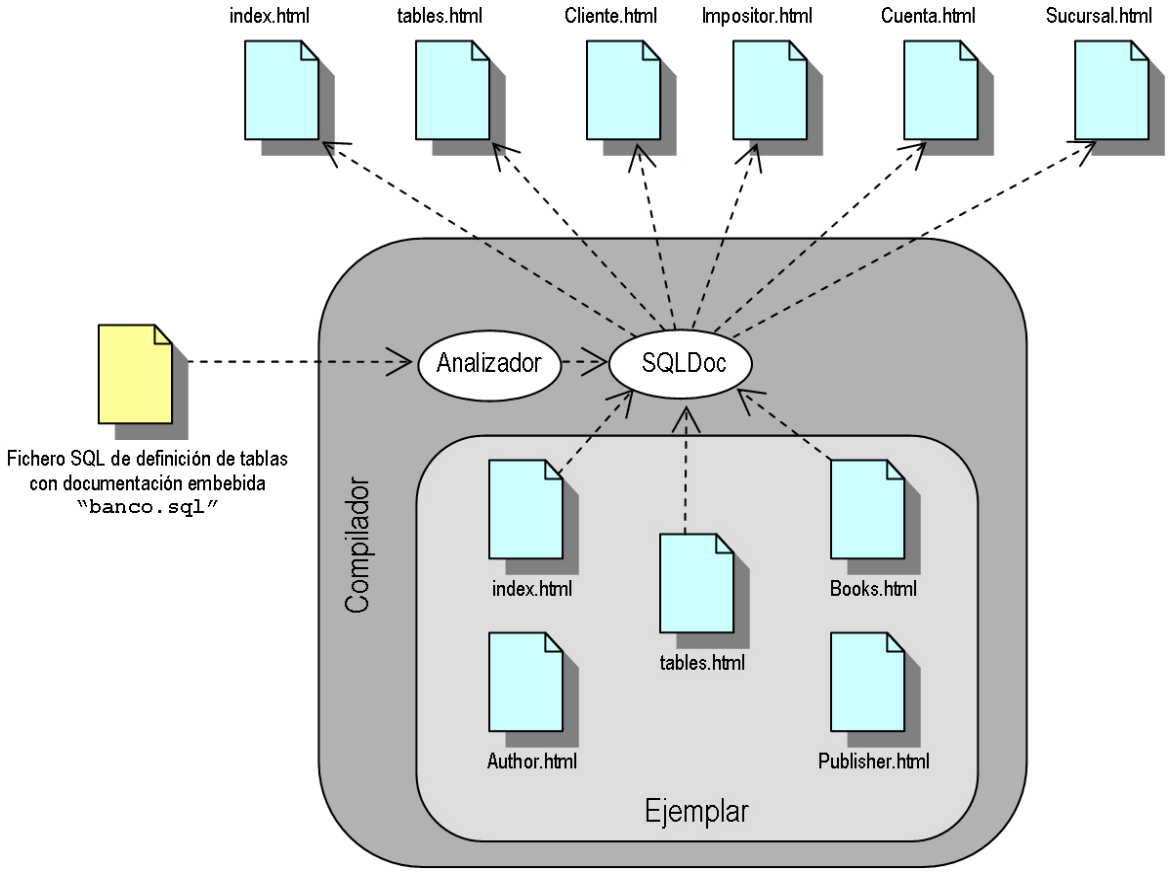


Figura 6.11. Arquitectura de SQLDoc.

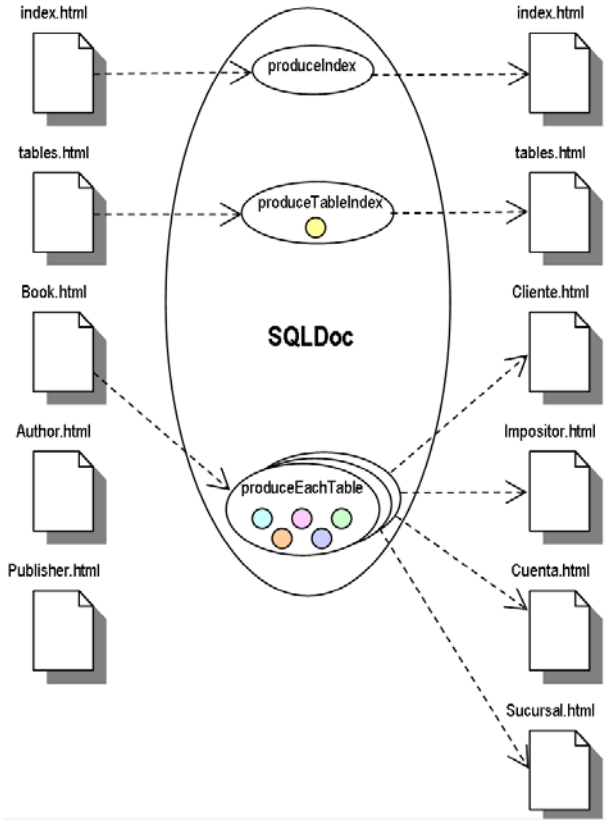


Figura 6.12. Diseño del generador SQLDoc.

```
##################################################
# Analyzer
##################################################

require "analyzer/Analyzer"

unless ARGV[0]
  print "SQLDoc usage: sqldoc sql_file\n"
  exit
end

fh = File.open(ARGV[0])
in_text = fh.read()
fh.close()

analyzer = Analyzer.new(in_text)

##################################################
# Generator
##################################################

require '..\..\ETL\ETL'

class SQLDoc < Generator

  def initialize(exemplar, output, analyzer)
    @exemplar=exemplar
    @output = output
    @analyzer = analyzer
    @tables = @analyzer.tables

    produceTableIndex
    produceEachTable
    prod("#{@exemplar}/index.html", "#{@output}/index.html", [], 'produceIndex')
  end
```

```
  def produceTableIndex
    tableRefRegExp = /((<a href=")\w+(.html" target="main">)\w+(<\/a><br>\s*))+/
    tableRefMatch = extract("#{@exemplar}/tables.html", tableRefRegExp)
    tableRefSub = ''
    @tables.each {|table|
      tableRefSub += tableRefMatch[2] + table.name +
                tableRefMatch[3] + table.name + tableRefMatch[4]
    }
    sub(tableRefRegExp, tableRefSub, 'tables')
    prod("#{@exemplar}/tables.html", "#{@output}/tables.html", ['tables'])
  end

  def produceEachTable
    @tables.each { |table|
      comment_analyzer = @analyzer.comment_analyzer(table.comment)
      table.fields.each { |field|
          field.comment = comment_analyzer.fieldDescriptions[ field.name.to_s.downcase.strip ]
      }
      table_comment = comment_analyzer.tableDescription
      relates_to = comment_analyzer.relatesTo

      sub(/(<title>).+?(<\/title>)/, '\1'+table.name+'\2',
        "tableTitle_#{table.name}")
      sub(/(Table Name:<\/td><td>).+?(<)/, '\1'+table.name+'\2',
        "tableName_#{table.name}")
      sub(/(Description:<\/td><td>).+?(<)/, '\1'+table_comment+'\2',
        "tableDescription_#{table.name}")

      fieldsRegExp = /<tr>\s*((<td\s+bgcolor.+<\/td>\s*){4,}<\/tr>(<tr>)?\s*)+/
      fieldsMatch = extract("#{@exemplar}/Book.html", fieldsRegExp)
      fieldMatch = /(<tr>\s*)((<td.+?>).+?(<\/td>\s*))*(<\/tr>)/.match(fieldsMatch[0])
      fieldsSub = ''
      table.fields.each { |field|
        constraints = []
        constraints.push( "Non-null" ) if ( field.not_null )
        constraints.push( "Unique" ) if ( field.unique )
        constraints.push( "Primary key" ) if ( field.primary_key )
        fieldsSub += fieldMatch[1] +
          fieldMatch[3] + field.name.to_s + fieldMatch[4] +
          fieldMatch[3] + field.type + fieldMatch[4] +
          fieldMatch[3] + constraints.join(", ") + fieldMatch[4] +
          fieldMatch[3] + field.comment + fieldMatch[4] +
          fieldMatch[5]
      }
      sub(fieldsRegExp, fieldsSub, "field_#{table.name}")

      relationRegExp = /(<td.*Relates To.*\s.*top>)((<a.+?>).+?(<\/a>),?\s*)+(<\/td>)/
      relationMatch = extract("#{@exemplar}/Book.html", relationRegExp)
      if (relates_to.length > 0)
        relationSub = relationMatch[1] + relates_to.collect { |t|
            "<a href=\"#{t}.html\">#{t}</a>"
          }.join(', ') + relationMatch[5]
      else
        relationSub = ''
      end
      sub(relationRegExp, relationSub, "relation_#{table.name}")

      prod("#{@exemplar}/Book.html", "#{@output}/#{table.name}.html",
        ["tableTitle_#{table.name}",
         "tableName_#{table.name}",
         "tableDescription_#{table.name}",
         "field_#{table.name}",
         "relation_#{table.name}"])
    }
  end

end

SQLDoc.new('exemplar', 'output', analyzer).gen
```

Figura 6.13. Código de SQLDoc (SQLDoc.rb).

6.2. Lectura de datos

Supóngase un programa escrito en **Java**, que maneja datos de diferentes personas almacenados en un fichero con formato CSV (*Comma Separated Value*) como el de la figura 6.14. Para recuperar la información, el programa tiene la clase *PeopleReader* de la figura 6.15, que lee los datos de cada persona y los guarda en un vector de objetos de la clase *People*.

```
"Pedro", 30
"Juan Antonio", 35
"Carmen", 29
```

Figura 6.14. Ejemplo de fichero de datos de personas ("data.csv").

```
import java.io.*;
import java.util.*;

/**
 * This is the base class for the CSV reader. You should derive your class from this
 * and make any modifications you like.
 */
public class PeopleReader {

  /**
   * The data structure class
   */
  public class  People
  {
    /**
    * The Name
    */
    public String name;

    /**
    * The Age
    */
    public Integer age;
  }

  private InputStream _in;
  private ArrayList _data;

  public PeopleReader( InputStream in )
  {
    _in = in;
    _data = new ArrayList();
  }

  /**
   * size returns the count of data rows found in the input.
   */
  public int size()
  {
    return _data.size();
  }

  /**
   * Returns the 'People' object for the specified row.
   */
  public People get( int index )
  {
    return (People) _data.get( index );
  }

  /**
```

```
 * Reads in the input stream
 */
public void read()
{
  StringBuffer line = new StringBuffer();
  line.setLength( 0 );

  try {
    while( true )
    {
      int nChar;
      if ( ( nChar = _in.read() ) == -1 ) {
        if ( line.length() > 0 )
                      process_line( line );
        break;
      }
      char cChar = (char)nChar;
      if ( cChar == '\n' )
      {
        process_line( line );
        line.setLength( 0 );
      }
      else
      {
        line.append( cChar );
      } // if
    } // while
  } // try
  catch( IOException except )
  {
    if ( line.length() > 0 )
      process_line( line );
  } // catch
} // read

/**
 * Processes a single input line.
 *
 * @arg line The row text
 */
private void process_line( StringBuffer line )
{
  ArrayList fields = new ArrayList();

  boolean inQuotes = false;
  boolean escaped = false;
  StringBuffer text = new StringBuffer();

  for( int index = 0; index < line.length(); index++ )
  {
    if ( escaped )
    {
      text.append( line.charAt( index ) );
    }
    else
    {
      if ( line.charAt( index ) == '\"' )
      {
        inQuotes = inQuotes ? false : true;
      }
      else if ( line.charAt( index ) == '\\' )
      {
        escaped = true;
      }
      else if ( line.charAt( index ) == ',' && ! inQuotes )
      {
        fields.add( new String( text ).trim() );
        text.setLength( 0 );
      }
      else
      {
        text.append( line.charAt( index ) );
      }
    }
```

```
    }

    if ( text.length() > 0 )
      fields.add( new String( text ).trim() );

    String strArray [] = new String[ fields.size() ];
    fields.toArray( strArray );

    process_fields( strArray );
  }

  /**
   * This is the main processor for a single row of string fields
   *
   * @arg fields The array of strings that make up the row.
   */
  private void process_fields( String fields[] )
  {
    People data = new People();
    data.name = fields[0];
    data.age = Integer.getInteger( fields[1] );
    _data.add( data );
  }

}
```

Figura 6.15. Lector de datos de personas ("PeopleReader.java").

Como muestra la figura 6.15, la lectura de ficheros de datos no es trivial. Por esta razón, y dado que es una tarea muy común, se han desarrollado soluciones generales como los meta-analizadores lex y yacc [BLM92] o los analizadores DOM (*Document Object Model*) y SAX (*Simple API for XML*) de XML [Har02].

En la presente sección se aplicará EDD y ETL para realizar, utilizando la figura 6.15 como ejemplar, una infraestructura de desarrollo automático de lectores de ficheros CSV.

6.2.1. Análisis de la familia de productos

Los lectores procesarán los datos de un fichero como el de la figura 6.14 y los almacenarán en un vector de objetos. Los objetos serán de una clase con tantos atributos como valores separados por comas tengan las líneas del fichero. Cada objeto se corresponderá con una línea del fichero.

La figura 6.16 es un diagrama FODA que modela el dominio de los "lectores de ficheros CSV".

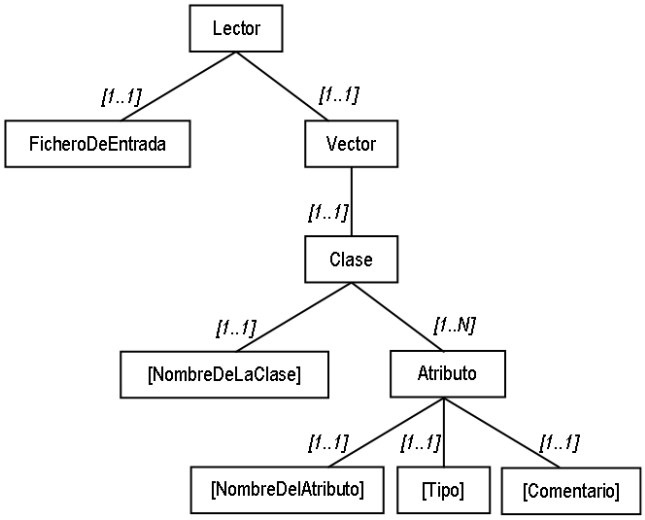
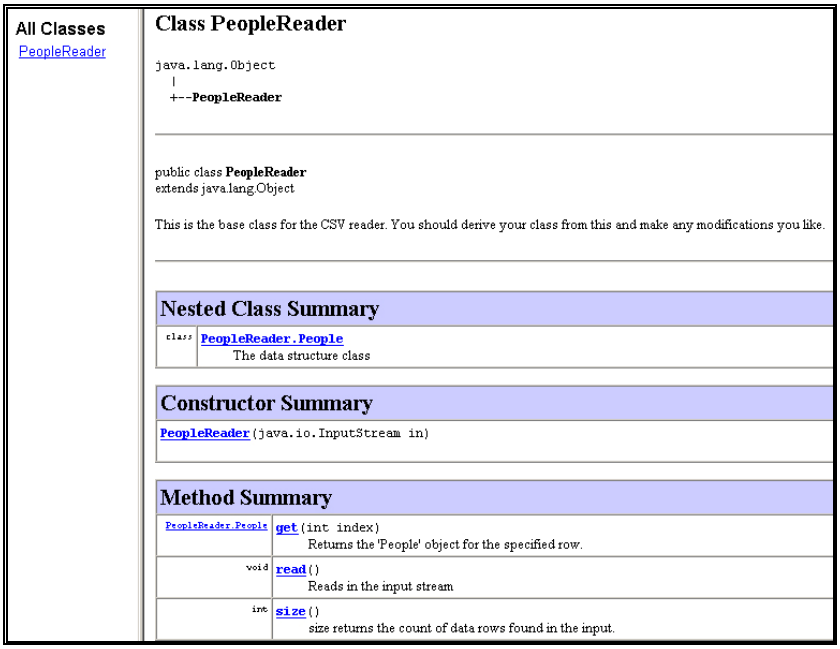


Figura 6.16. Modelo del dominio "lectores de ficheros CSV"

La herramienta producirá, además de los lectores,

- **documentación**, que, como en el ejemplar (en tinta azul), estará embebida en el código. Posteriormente, con la herramienta *Javadoc* se generará automáticamente un conjunto de ficheros HTML que facilitarán la consulta de la documentación (figura 6.17).

- un **juego de pruebas**. Para su generación se utilizará como ejemplar el juego de pruebas de la figura 6.19, que aprovecha el marco de trabajo *JUnit* [JUn06] para validar la lectura del fichero de la figura 6.14.



All Classes
PeopleReader

Class PeopleReader

```
java.lang.Object
  |
  +--PeopleReader
```

public class **PeopleReader**
extends java.lang.Object

This is the base class for the CSV reader. You should derive your class from this and make any modifications you like.

Nested Class Summary

class	**PeopleReader.People** The data structure class

Constructor Summary

PeopleReader(java.io.InputStream in)

Method Summary

PeopleReader.People	**get**(int index) Returns the 'People' object for the specified row.
void	**read**() Reads in the input stream
int	**size**() size returns the count of data rows found in the input.

Figura 6.17. Documentación, generada con la herramienta javadoc, del lector de personas.

```
import junit.framework.*;
import java.io.*;

public class PeopleTest extends TestCase {

  private FileInputStream in;
  private PeopleReader reader;

  public void setUp() {

    try
    {
      in = new FileInputStream( "data.csv" );

      reader = new PeopleReader( in );
      reader.read();

      in.close();

    } catch( Exception e ) {
      System.err.println(e);
    } // catch

  } // setUp

  public void test1() {
    Assert.assertEquals(reader.get(0).name, "Pedro");
    Assert.assertEquals(reader.get(0).age, Integer.getInteger("30"));
  } // test1

  public void test2() {
    Assert.assertEquals(reader.get(1).name, "Juan Antonio");
    Assert.assertEquals(reader.get(1).age, Integer.getInteger("35"));
  } // test2

  public void test3() {
    Assert.assertEquals(reader.get(2).name, "Carmen");
    Assert.assertEquals(reader.get(2).age, Integer.getInteger("29"));
  } // test3

  public static Test suite() {
    return new TestSuite(PeopleTest.class);
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

Figura 6.18. Juego de pruebas para el lector de personas ("PeopleTest.java").

6.2.2. Definición de una interfaz para la especificación abstracta de los productos

Aplicando el proceso descrito en la sección 3.2.2.1.1 a la figura 6.16, se podará el nodo fijo *FicheroDeEntrada* y se obtendrá la gramática de la figura 6.19. Para simplificar el análisis de las especificaciones de los lectores, se utilizará una notación XML. Por ejemplo, la figura 6.20 es la especificación XML de un lector de libros y la figura 6.21, un fichero de datos sobre libros.

```
Lector ::= Vector;
```

```
Vector ::= Clase;
Clase ::= NombreDeLaClase {Atributo};
NombreDeLaClase ::= '[a-zA-Z]+';
Atributo ::= NombreDelAtributo Tipo Comentario;
NombreDelAtributo ::= '[a-zA-Z]+';
Tipo ::= '[a-zA-Z]+';
Comentario ::= '[a-zA-Z ]+';
```

Figura 6.19. Gramática EBNF del DSL con el que se especificarán los lectores de ficheros CSV.

```
<spec classname="Book">
  <field name="title" type="String" comment="The title" />
  <field name="author" type="String" comment="The author" />
  <field name="size" type="Integer" comment="The number of pages" />
  <field name="price" type="Float" comment="The price in dollars" />
</spec>
```

Figura 6.20. Especificación XML de un lector de libros.

```
"Programming Ruby: The Pragmatic Programmers' Guide", "Dave Thomas, Andy Hunt", 564, 31.40
"The Ruby Way", "Hal Fulton", 600, 27.99
```

Figura 6.21. Ejemplo de fichero de datos para el lector especificado en la figura 6.20.

6.2.3. Implementación de la flexibilización del ejemplar

La figura 6.22 representa el compilador que:

- recibirá la especificación XML de un lector y un ejemplo de fichero de datos CSV.
- producirá el programa lector con su documentación embebida (*NewReader.java*) y un juego de pruebas sobre el ejemplo de fichero de datos (*NewTest.java*).

La figura 6.23 es el analizador que procesa las especificaciones XML de los lectores aprovechando la librería REXML [Rus06]. El resto del análisis se resolverá mediante expresiones regulares.

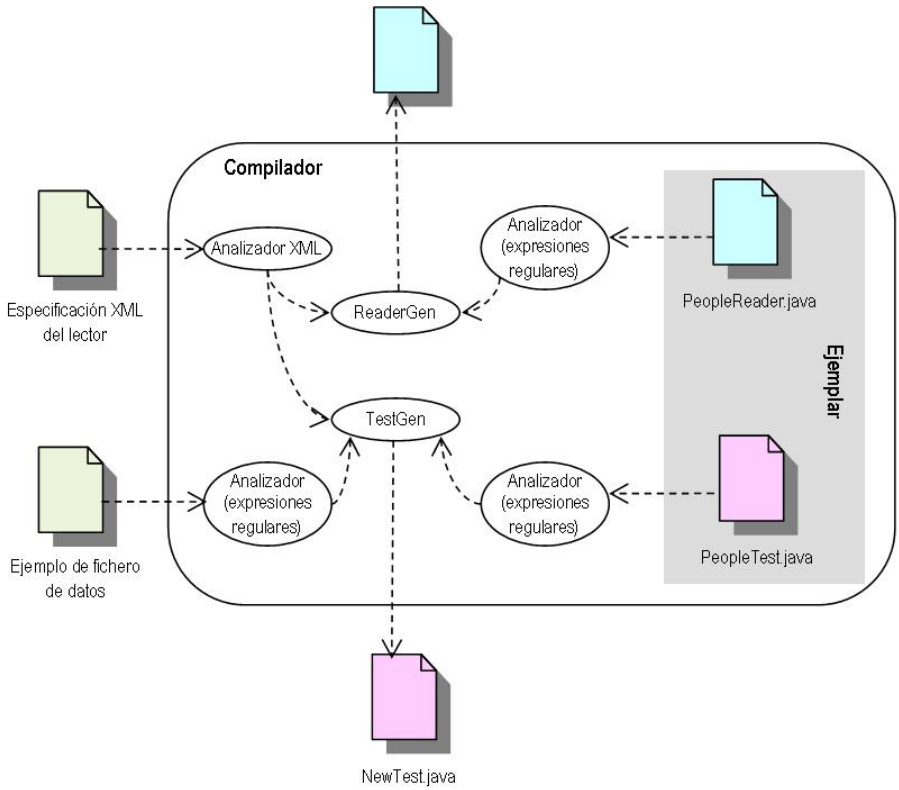


Figura 6.22. Diseño del compilador de "lectores de ficheros CSV".

```
def read_csv_grammar(fileName, reader)
  doc = REXML::Document.new(File.open(fileName))
  reader.classname = doc.root.attributes["classname"].to_s
  doc.root.each_element("field") { |fieldNode|
    field = Field.new()
    field.name = fieldNode.attributes["name"].to_s
    field.type = fieldNode.attributes["type"].to_s
    field.comment = fieldNode.attributes["comment"].to_s
    reader.fields.push(field)
  }
end

READER = 'PeopleReader.java'
TEST = 'PeopleTest.java'

unless ARGV[0] && ARGV[1] && ARGV[2]
  print "csvgen usage: csvgen csv_grammar csv_example out_dir\n"
  exit
end

xml_csv_grammar = ARGV[0]
csv_example = ARGV[1]
out_dir = ARGV[2]

Field = Struct.new("Field", :name, :type, :comment)
CSV_Grammar = Struct.new("CSV_Grammar", :classname, :fields)
csv_grammar = CSV_Grammar.new()
csv_grammar.fields = []

read_csv_grammar(xml_csv_grammar, csv_grammar)
```

Figura 6.23. Analizador de las especificaciones de programas lectores de datos.

Como se aprecia en la figura 6.22, para flexibilizar el ejemplar se emplearán dos generadores:

- *ReaderGen* (figura 6.25) gestionará la variabilidad de los lectores y de su documentación.

- *TestGen* (figura 6.26) gestionará la variabilidad de los juegos de pruebas de los lectores.

La figura 6.24 muestra la actuación de los generadores para obtener el programa objeto especificado en la figura 6.20. Se han utilizado los colores □, □, □, □ y □ para identificar el código variable en el ejemplar (figuras 6.15 y 6.18) y las sustituciones que lo adaptarán (figuras 6.24, 6.25 y 6.26).

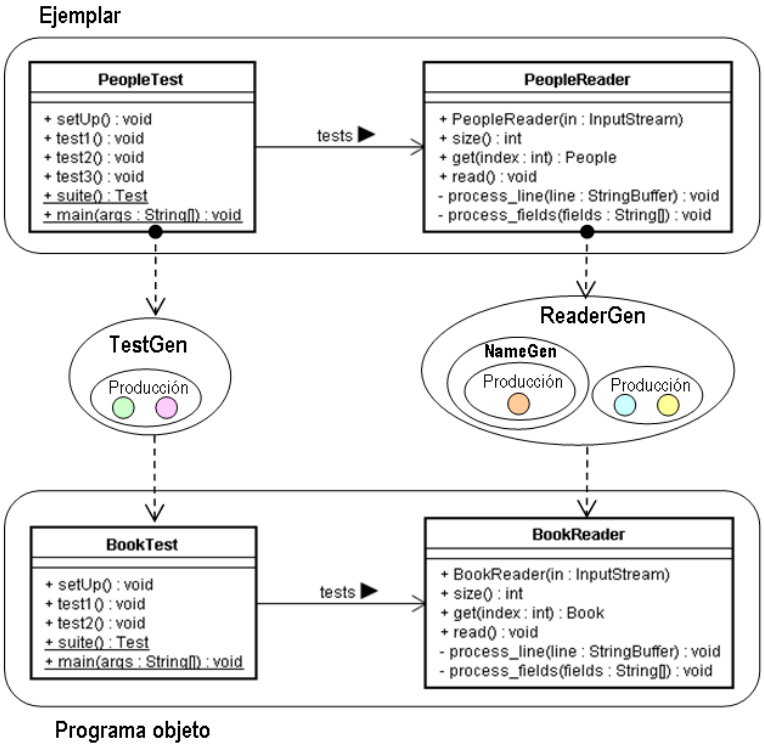


Figura 6.24. Ejemplo de generación del programa objeto especificado en la figura 6.20.

```
class ReaderGen < Generator
  include CSV_Utilities

  class NameGen < Generator
    def initialize(csv_grammar, reader, out_dir)
      gsub(/People/, csv_grammar.classname)
      prod(reader, "#{out_dir}/#{csv_grammar.classname}Reader.java")
    end
  end

  def initialize(csv_grammar, reader, out_dir)
    NameGen.new(csv_grammar, reader, out_dir).gen
```

```
    dataStructureClassRegExp = /
      public\s+class\s+#{csv_grammar.classname}\s+\{
        .*?
      \}
    /xm

    sub(dataStructureClassRegExp, dataStructureClassTemplate(csv_grammar))

    sub(/(\s*\bdata\..+?;)+/, process_fields_template(csv_grammar.fields))

    prod("#{out_dir}/#{csv_grammar.classname}Reader.java",
      "#{out_dir}/#{csv_grammar.classname}Reader.java")
  end

  def dataStructureClassTemplate(csv_grammar)
    fieldsCode = ''
    csv_grammar.fields.each { |field|
      fieldsCode += dataStructureFieldTempate(field)
    }
    code = <<-END_OF_CODE
      public class #{csv_grammar.classname}
      {
      #{fieldsCode}
      }
    END_OF_CODE
  end

  def dataStructureFieldTempate(field)
    code = <<-END_OF_CODE
      /**
        * #{field.comment}
        */
       public #{field.type} #{field.name};
    END_OF_CODE
  end

  def process_fields_template(fields)
    code = ''
    fields.each_index { |i|
      rigth_side = proccessField(fields[i].type, "fields[#{i}]")
      code += "data.#{fields[i].name} = #{rigth_side};\n"
    }
    return code
  end
end
```

Figura 6.25. Generador ReaderGen.

```
class TestGen < Generator
  include CSV_Utilities

  def initialize(csv_grammar, csv_example, test, out_dir)

    gsub(/People/, csv_grammar.classname)

    testsRegExp = /
       (
      public\s+\w+\s+test.+?\{
        .*?
      \}\s*\/\/\s*\w+\s*
       )+
    /xm
    testsSub = ''
    i = 1
    File.open(csv_example).each_line{ |line|
      csvRegExp = /
        (?:^|,)\s*
        (
          "[^"]*"
          |
          [^",]*
```

```
        )
      /x
      testData = line.scan(csvRegExp).flatten
      testData.collect {|data|
        data.tr!('"', '')
        data.strip!
      }
      testsSub += testTemplate(i, testData, csv_grammar)
      i += 1
    }
    sub(testsRegExp, testsSub)

    prod(test, "#{out_dir}/#{csv_grammar.classname}Test.java")
  end

  def assertTemplate(testIndex, field, value)
    value = "\"#{value}\""
    code = "Assert.assertEquals(reader.get(#{testIndex-1})." +
         "#{field.name}, #{proccessField(field.type, value)});\n"
  end

  def testTemplate(testIndex, testData, csv_grammar)
    assertCode = ''
    csv_grammar.fields.each_index { |i|
      assertCode += assertTemplate(testIndex, csv_grammar.fields[i], testData[i])
    }

    code = "public void test#{testIndex}() {\n" +
        assertCode +
        "} // test#{testIndex}\n\n"
  end
end
```

Figura 6.26. Generador TestGen.

La figura 6.27 es el módulo auxiliar *CSV_Utilities* utilizado por *ReaderGen* y *TestGen*.

```
module CSV_Utilities

  def proccessField(type, value)
    case type
      when 'Boolean', 'Byte', 'Double', 'Float', 'Integer', 'Long', 'Short'
        "#{type}.valueOf(#{value})"
      else
        value
    end
  end

end
```

Figura 6.27. Módulo auxiliar CSV_Utilities.

Como puede apreciarse en la figura 6.15, entre las sustituciones □ y □ se produciría una colisión (`People`). Para evitarla, el generador *ReaderGen* aplicará en primer lugar la sustitución O mediante su generador auxiliar *NameGen*.

Finalmente, la figura 6.28 es el código que coordina la acción de *ReaderGen* y *TestGen*.

```
(
  ReaderGen.new(csv_grammar, READER, out_dir) +
  TestGen.new(csv_grammar, csv_example, TEST, out_dir)
).gen
```

Figura 6.28. Suma de los generadores ReaderGen y TestGen.

6.3. Copia de seguridad de una base de datos

Una organización, que trabaja con grandes cantidades de información almacenada en una base de datos relacional, desearía realizar periódicamente copias de seguridad, guardando los datos en tablas temporales. La figura 6.29 muestra algunas de las tablas de la base de datos.

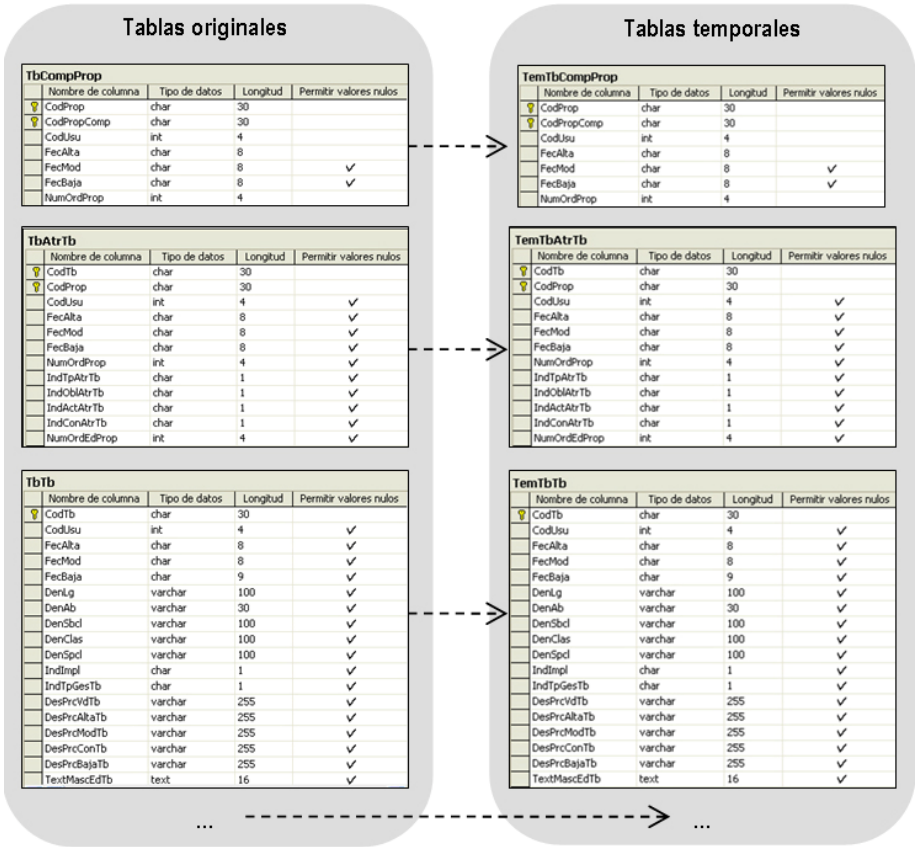


Figura 6.29. Copia de datos entre tablas originales y temporales.

Para garantizar la fiabilidad de los datos y facilitar la depuración de los errores, la organización estima conveniente registrar cada copia en una tabla de seguimiento o "de *log*" llamada *TbDiaCTlPrc* (figura 6.30).

Para automatizar la copias de seguridad, el departamento de informática de la organización comienza desarrollando los procedimientos almacenados

sp_AprovTbCompProp, sp_ActuTbCompProp y *sp_ExisTbCompProp*, que aprovisionan la tabla temporal *TemTbCompProp* con los datos de la tabla *TbCompProp*, y el procedimiento almacenado *sp_ActuTbDiaCtlPrc*, que registra las copias en la tabla de seguimiento. La figura 6.31 muestra mediante un diagrama de abstracciones los citados procedimientos.

TbDiaCtlPrc

	Nombre de columna	Tipo de datos	Longitud	Permitir valores nulos
🔑	TpoRl	datetime	8	
🔑	CodPrc	char	30	
	CodUsu	int	4	✓
	CodEvt	char	30	✓
🔑	NumEvt	int	4	
	IdObjEvt	varchar	255	✓
	DesObjEvt	text	16	✓
	ExprObjEvt	text	16	✓

Figura 6.30. Tabla de seguimiento de la copia de datos.

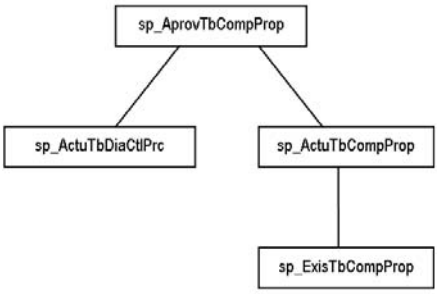


Figura 6.31. Procedimientos almacenados para la copia de los datos contenidos en la tabla TbCompProp.

Las figuras 6.32, 6.33, 6.34 y 6.35 son respectivamente los códigos de los procedimientos almacenados *sp_AprovTbCompProp, sp_ActuTbDiaCtlPrc, sp_ActuTbCompProp* y *sp_ExisTbCompProp* (los procedimientos están escritos en el lenguaje **TRANSACT SQL** , puesto que la organización utiliza el gestor de base de datos *Microsoft SQL Server).*

```
/*  sp_AprovTbCompProp  */
/*----------------------*/

/*  Actualizacion Masiva de Tabla TbCompProp.  */

IF EXISTS (SELECT * FROM sysobjects WHERE
   id = object_id('sp_AprovTbCompProp') AND
   sysstat & 0xf = 4)
   DROP PROCEDURE sp_AprovTbCompProp

GO

CREATE PROCEDURE sp_AprovTbCompProp
/*                                                    */
/**  DECLARACION DE PARAMETROS DEL PROCEDIMIENTO.  **/

@P_CodUsu int ,
@P_Actualizacion int = 1  ,
@P_Validacion int = 0  ,
@P_Referencial int = 0  ,
@P_Transformacion int = 1  ,
@P_Mensajes int = 2  ,
@P_NumRegLOG int = -1  ,
@P_NumRegDiaCtlPrc int = 5000  ,
@P_Error  int OUTPUT

AS
/*                                                    */
/**  DECLARACION DE VARIABLES LOCALES DEL PROCEDIMIENTO.  **/
/*                                                    */
/**  Registros locales para tratar la tabla BASE.  **/

DECLARE @L_CodProp char (30)
DECLARE @L_CodPropComp char (30)
DECLARE @L_CodUsu int
```

```
DECLARE @L_FecAlta char (8)
DECLARE @L_FecMod char (8)
DECLARE @L_FecBaja char (8)
DECLARE @L_NumOrdProp int
/*                                                                  */
/**  Otras variables locales del procedimiento.  **/

DECLARE @L_NumFilasTotal int
DECLARE @L_NumFilasTratadas int
DECLARE @L_NumFilasError int
DECLARE @L_NumFilasActu int
DECLARE @L_NumFilasAlta int
DECLARE @L_NumFilasModif int
DECLARE @L_ErrorActu int
DECLARE @L_Inicio datetime
DECLARE @L_UltimaVez datetime
DECLARE @L_RegAlta int
DECLARE @L_RegModif int
DECLARE @L_CursorAbierto int
DECLARE @msg varchar(255)
DECLARE @msg_Log varchar(255)
DECLARE @L_UltClaveActu varchar(255)
DECLARE @L_NumEvt int
DECLARE @L_NombreBD char(30)
/*                                                                  */
/**  INICIO DEL PROCEDIMIENTO.  **/

IF (@P_NumRegLOG > 0) BEGIN TRANSACTION

SET NOCOUNT ON

SELECT @P_Error = 0
SELECT @L_ErrorActu = 0
SELECT @msg = ''
SELECT @msg_Log = ''

SELECT @L_NumFilasTotal = 0
SELECT @L_NumFilasTratadas = 0
SELECT @L_NumFilasError = 0
SELECT @L_NumFilasActu = 0
SELECT @L_NumFilasAlta = 0
SELECT @L_NumFilasModif = 0

SELECT @L_Inicio = GetDate()
SELECT @L_CursorAbierto = 0
SELECT @L_UltClaveActu = ''
SELECT @L_NombreBD = DB_NAME()

IF @P_Mensajes > 0
   BEGIN
   SELECT 'Inicio' = CONVERT(varchar, @L_Inicio ,113)
   PRINT ''
   END
/*                                                                  */
/**  Acceso a la tabla TEMPORAL de datos.  **/

DECLARE rs SCROLL CURSOR FOR

SELECT
  CodProp  ,
  CodPropComp  ,
  CodUsu  ,
  FecAlta  ,
  FecMod  ,
  FecBaja  ,
  NumOrdProp

FROM TemTbCompProp

ORDER BY
      CodProp  ,
      CodPropComp

IF @@ERROR <> 0
```

```
   BEGIN
   SELECT  @msg  = 'ERROR de acceso a tabla TEMPORAL.'
   GOTO Err_Aprov
   END

IF @P_Mensajes = 1 PRINT 'Apertura del CURSOR.'

OPEN rs

IF @@ERROR <> 0
   BEGIN
   SELECT  @msg  = 'ERROR de apertura del CURSOR.'
   GOTO Err_Aprov
   END

SELECT @L_CursorAbierto = 1

SELECT @L_NumFilasTotal = @@CURSOR_ROWS

IF @P_Mensajes > 0
   BEGIN
   SELECT 'Tiempo Apertura Cursor' = DateDiff(ms, @L_Inicio ,GetDate()) , 'Num.Filas.Cursor'
=  @L_NumFilasTotal
   PRINT ''
   END

SELECT @L_UltimaVez = GetDate()

FETCH NEXT FROM rs INTO
@L_CodProp  ,
@L_CodPropComp  ,
@L_CodUsu  ,
@L_FecAlta  ,
@L_FecMod  ,
@L_FecBaja  ,
@L_NumOrdProp

IF @@ERROR <> 0
   BEGIN
   SELECT  @msg  = 'ERROR al recuperar informacion del CURSOR.'
   GOTO Err_Aprov
   END

SELECT @msg_Log = 'INIPROC;NumFilasTotal=' + CONVERT(varchar,@L_NumFilasTotal)

SELECT @L_NumEvt = 0
EXEC sp_ActuTbDiaCtlPrc
     @P_CodPrc = 'sp_AprovTbCompProp' ,
     @P_CodUsu =  @P_CodUsu  ,
     @P_CodEvt =  'INICIO' ,
     @P_NumEvt =  @L_NumEvt  OUTPUT ,
     @P_IdObjEvt =  @msg_Log

WHILE @@FETCH_STATUS <> -1

BEGIN

IF (@P_NumRegLOG > 0) AND (@L_NumFilasTratadas % @P_NumRegLOG = 0) AND (@L_NumFilasTratadas
<> 0)

BEGIN /* Cada NumRegLOG Registros: */

COMMIT

DUMP TRANSACTION @L_NombreBD WITH TRUNCATE_ONLY

IF @@ERROR <> 0
   BEGIN
   SELECT  @msg  = 'ERROR al truncar el LOG de transacciones.'
   GOTO Err_Aprov
   END

IF @P_Mensajes = 1
   BEGIN
```

```
   PRINT 'LOG de transacciones TRUNCADO correctamente.'
   PRINT ''
   END

BEGIN TRANSACTION

END /* FIN de Cada NumRegLOG Registros */
/*                                                                    */
/**  Anotar en LOG PROCESOS el ultimo registro actualizado.  **/

IF  (@P_NumRegDiaCtlPrc  >  0)  AND  (@L_NumFilasTratadas  <>  0)  AND  (@L_NumFilasTratadas  %
@P_NumRegDiaCtlPrc = 0)
   BEGIN

   SELECT  @msg_Log  =  'NumFilas='  +  CONVERT(varchar,@L_NumFilasActu)  +  ';UltClave='  +
@L_UltClaveActu

   EXEC sp_ActuTbDiaCtlPrc
        @P_CodPrc = 'sp_AprovTbCompProp' ,
        @P_CodUsu =  @P_CodUsu  ,
        @P_CodEvt =  '' ,
        @P_NumEvt =  @L_NumEvt  ,
        @P_IdObjEvt =  @msg_Log

   IF @P_Mensajes > 0
      BEGIN
      SELECT
        'TOTAL Filas' =  @L_NumFilasTotal  ,
        'Filas TRATADAS' =  @L_NumFilasTratadas  ,
        'Tiempo' = DateDiff(ms, @L_UltimaVez ,GetDate()) ,
        'Tiempo Total' = DateDiff(ms, @L_Inicio ,GetDate()) ,
        'Filas Actu' =  @L_NumFilasActu  ,
        'Filas Alta' =  @L_NumFilasAlta  ,
        'Filas Modif' =  @L_NumFilasModif  ,
        'Filas Error' =  @L_NumFilasError
      PRINT ''
      END

   SELECT @L_UltimaVez = GetDate()

   END /* FIN de cada NumRegDiaCtlPrc registros. */
/*                                                                    */
/**  ACTUALIZACION de elementos de la estructura de datos.  **/

SELECT @L_RegAlta = 0
SELECT @L_RegModif = 0
SELECT @L_ErrorActu = 0

EXEC sp_ActuTbCompProp
     @L_CodProp  ,
     @L_CodPropComp  ,
     @P_CodUsu  ,
     @L_FecAlta  ,
     @L_FecMod  ,
     @L_FecBaja  ,
     @L_NumOrdProp  ,
     @P_Mensajes  = @P_Mensajes  ,
     @P_Actualizacion  = @P_Actualizacion  ,
     @P_Validacion  = @P_Validacion  ,
     @P_Referencial  = @P_Referencial  ,
     @P_Transformacion  = @P_Transformacion  ,
     @P_RegCreado  =  @L_RegAlta  OUTPUT ,
     @P_RegModificado  =  @L_RegModif  OUTPUT ,
     @P_Error  =  @L_ErrorActu  OUTPUT

IF (@@ERROR <> 0) OR (@L_ErrorActu = 1)
   BEGIN
   SELECT  @msg  = 'ERROR en ACTUALIZACION de elementos.'
   GOTO Err_Aprov
   END

SELECT @L_UltClaveActu =
       @L_CodProp  + '' +
       @L_CodPropComp
```

```
IF @L_RegAlta = 1
   SELECT  @L_NumFilasAlta  =  @L_NumFilasAlta  + 1

IF @L_RegModif = 1
   SELECT  @L_NumFilasModif  =  @L_NumFilasModif  + 1

IF (@L_RegAlta = 1) OR (@L_RegModif = 1)
   SELECT  @L_NumFilasActu  =  @L_NumFilasActu  + 1

SELECT @L_NumFilasTratadas = @L_NumFilasTratadas + 1

FETCH NEXT FROM rs INTO
@L_CodProp  ,
@L_CodPropComp  ,
@L_CodUsu  ,
@L_FecAlta  ,
@L_FecMod  ,
@L_FecBaja  ,
@L_NumOrdProp

IF @@ERROR <> 0
   BEGIN
   SELECT  @msg  = 'ERROR en recuperacion del CURSOR.'
   GOTO Err_Aprov
   END

END /* Fin de WHILE. */

CLOSE rs

DEALLOCATE rs

SELECT @L_CursorAbierto = 0

IF @P_Mensajes = 1
   BEGIN
   PRINT 'FIN del tratamiento de registros.'
   PRINT ''
   END
/*                                                                  */
/**  FIN DEL PROCEDIMIENTO.  **/

IF (@P_NumRegLOG > 0) AND (@@TRANCOUNT > 0)
   IF  @P_Error  = 0 COMMIT
   ELSE ROLLBACK

SELECT @msg_Log = 'FINPROC' +
  ';TOTALFilas=' + CONVERT(varchar, @L_NumFilasTotal ) +
/*  ';FilasTRATADAS=' + CONVERT(varchar, @L_NumFilasTratadas ) + */
  ';TiempoTotal=' + CONVERT(varchar,DateDiff(ms, @L_Inicio ,GetDate())) +
/*  ';FilasActu=' + CONVERT(varchar, @L_NumFilasActu ) + */
  ';FilasAlta=' + CONVERT(varchar, @L_NumFilasAlta ) +
  ';FilasModif=' + CONVERT(varchar, @L_NumFilasModif ) +
/*  ';FilasError=' + CONVERT(varchar, @L_NumFilasError ) + */
  ';ErrorSalida=' + CONVERT(varchar, @P_Error )

EXEC sp_ActuTbDiaCtlPrc
     @P_CodPrc = 'sp_AprovTbCompProp' ,
     @P_CodUsu =  @P_CodUsu  ,
     @P_CodEvt =  'FIN' ,
     @P_NumEvt =  @L_NumEvt  ,
     @P_IdObjEvt =  @msg_Log

IF @P_Mensajes > 0
   BEGIN
   SELECT
     'TOTAL Filas' =  @L_NumFilasTotal  ,
     'Filas TRATADAS' =  @L_NumFilasTratadas  ,
     'Tiempo' = DateDiff(ms, @L_UltimaVez ,GetDate()) ,
     'Tiempo Total' = DateDiff(ms, @L_Inicio ,GetDate()) ,
     'Filas Actu' =  @L_NumFilasActu  ,
     'Filas Alta' =  @L_NumFilasAlta  ,
     'Filas Modif' =  @L_NumFilasModif  ,
```

```
     'Filas Error' =  @L_NumFilasError  ,
     'ErrorSalida' =  @P_Error
   PRINT ''
   END

SET NOCOUNT OFF

RETURN
/*                                                                  */
/**  Tratamiento de ERRORES.  **/

Err_Aprov:

SELECT @P_Error = 1

IF @L_CursorAbierto = 1 DEALLOCATE rs

IF (@P_NumRegLOG > 0) AND (@@TRANCOUNT > 0) ROLLBACK

SELECT @msg_Log = 'FINPROC'
  + ';ErrorSalida=' + CONVERT(varchar, @P_Error )
  + ';MsgError=' +  @msg

EXEC sp_ActuTbDiaCtlPrc
     @P_CodPrc = 'sp_AprovTbCompProp' ,
     @P_CodUsu = @P_CodUsu ,
     @P_CodEvt = 'FIN' ,
     @P_NumEvt = @L_NumEvt ,
     @P_IdObjEvt = @msg_Log

RAISERROR (@msg,16,1)

SET NOCOUNT OFF

RETURN

GO
```

Figura 6.32. Procedimiento almacenado sp_AprovTbCompProp.

```
/*  sp_ActuTbDiaCtlPrc  */
/*----------------------*/

/*  Actualizacion de Tabla TbDiaCtlPrc.  */

IF EXISTS (SELECT * FROM sysobjects WHERE
   id = object_id('sp_ActuTbDiaCtlPrc') AND
   sysstat & 0xf = 4)
   DROP PROCEDURE sp_ActuTbDiaCtlPrc

GO

CREATE PROCEDURE sp_ActuTbDiaCtlPrc
/*                                                                  */
/**  DECLARACION DE PARAMETROS DEL PROCEDIMIENTO.  **/

@P_CodPrc char(30) ,
@P_CodUsu int  ,
@P_CodEvt char(30)  = '' ,
@P_NumEvt int  = 0 OUTPUT ,
@P_IdObjEvt varchar(255)  = '' ,
@P_DesObjEvt text  = '' ,
@P_ExprObjEvt text  = ''

AS
/*                                                                  */
/**  INICIO DEL PROCEDIMIENTO.  **/
/*                                                                  */
/**  Calculo del numero siguiente de evento para el proceso.  **/

IF @P_NumEvt = 0
```

```
   BEGIN
   SELECT  @P_NumEvt  = MAX(NumEvt)
   FROM  TbDiaCtlPrc
   WHERE CodPrc =  @P_CodPrc
   IF  @P_NumEvt  IS NULL
   SELECT  @P_NumEvt  = 1
   ELSE
   SELECT  @P_NumEvt  =  @P_NumEvt  + 1
   END
/*                                                                  */
/**  Actualizacion del Diario de Control de Procesos.  **/

INSERT INTO TbDiaCtlPrc (
  TpoRl  ,
  CodPrc  ,
  CodUsu  ,
  CodEvt  ,
  NumEvt  ,
  IdObjEvt  ,
  DesObjEvt  ,
  ExprObjEvt  )

VALUES (
  GetDate()  ,
  @P_CodPrc  ,
  @P_CodUsu  ,
  @P_CodEvt  ,
  @P_NumEvt  ,
  @P_IdObjEvt  ,
  @P_DesObjEvt  ,
  @P_ExprObjEvt  )
/*                                                                  */
/**  FIN DEL PROCEDIMIENTO.  **/

GO
```

Figura 6.33. Procedimiento almacenado sp_ActuTbDiaCtlPrc.

```
/*  sp_ActuTbCompProp  */
/*---------------------*/

/*  Actualizacion Puntual de Tabla TbCompProp.  */

IF EXISTS (SELECT * FROM sysobjects WHERE
   id = object_id('sp_ActuTbCompProp') AND
   sysstat & 0xf = 4)
   DROP PROCEDURE sp_ActuTbCompProp

GO

CREATE PROCEDURE sp_ActuTbCompProp
/*                                                                  */
/**  DECLARACION DE PARAMETROS DEL PROCEDIMIENTO.  **/
/*                                                                  */
/**  Identificadores de la estructura.  **/

@P_CodProp char (30) ,
@P_CodPropComp char (30) ,
/*                                                                  */
/**  Descriptores de la estructura.  **/

@P_CodUsu int  ,
@P_FecAlta char (8) ,
@P_FecMod char (8) ,
@P_FecBaja char (8) ,
@P_NumOrdProp int  ,
/*                                                                  */
/**  Otros parametros del procedimiento.  **/

@P_ExisteRegEnBase int = -1 ,
@P_Mensajes int = 2  ,
```

```
@P_Actualizacion int = 1  ,
@P_Validacion int = 0  ,
@P_Referencial int = 1  ,
@P_Transformacion int = 1  ,
@P_RegCreado int  OUTPUT,
@P_RegModificado int  OUTPUT,
@P_Error int  OUTPUT

AS
/*                                                              */
/**  DECLARACION DE VARIABLES LOCALES DEL PROCEDIMIENTO.  **/

DECLARE @L_Hoy char(8)
DECLARE @L_FecAlta char(8)
DECLARE @L_FecMod char(8)
DECLARE @L_FecBaja char(8)
DECLARE @L_ErrorExistencia int
DECLARE @L_ErrorTransformacion int
DECLARE @msg varchar(255)
/*                                                              */
/**  INICIO DEL PROCEDIMIENTO.  **/

SELECT @P_Error = 0
SELECT @L_ErrorExistencia = 0
SELECT @L_ErrorTransformacion = 0
SELECT @P_RegCreado = 0
SELECT @P_RegModificado = 0

SELECT @L_Hoy = CONVERT(char(8),GetDate(),112)

SELECT @L_FecAlta = ''
SELECT @L_FecMod = ''
SELECT @L_FecBaja = ''
/*                                                              */
/**  Comprobacion existencia de elemento en la estructura de datos.  **/

IF @P_ExisteRegEnBase = -1

BEGIN

EXEC @P_ExisteRegEnBase = sp_ExisTbCompProp
     @P_CodProp  ,
     @P_CodPropComp  ,
     @L_ErrorExistencia  OUTPUT

IF (@@ERROR <> 0) OR (@L_ErrorExistencia = 1)
   BEGIN
   SELECT  @msg  = 'ERROR de comprobacion de existencia en  TbCompProp .'
   GOTO Err_Act
   END

END
/*                                                              */
/**  Actualizacion de la estructura de datos.  **/

IF @P_Actualizacion = 1

BEGIN

IF @P_ExisteRegEnBase = 0

   BEGIN

   SELECT @L_FecAlta = @L_Hoy

   INSERT INTO TbCompProp (
     CodProp  ,
     CodPropComp  ,
     CodUsu  ,
     FecAlta  ,
     FecMod  ,
     FecBaja  ,
     NumOrdProp  )
```

```
   VALUES (
     @P_CodProp  ,
     @P_CodPropComp  ,
     @P_CodUsu  ,
     @L_FecAlta  ,
     @L_FecMod  ,
     @L_FecBaja  ,
     @P_NumOrdProp  )

   SELECT @P_Error = @@ERROR , @P_RegCreado = @@ROWCOUNT

   IF @P_Error <> 0
      BEGIN
      SELECT  @msg  = 'ERROR de ALTA en  TbCompProp .'
      GOTO Err_Act
      END

   END /* Fin IF ExisteRegEnBase. */

ELSE /* ExisteRegEnBase = 1. */

   BEGIN

   SELECT @L_FecMod = @L_Hoy

   UPDATE TbCompProp
   SET
     CodUsu  =  @P_CodUsu  ,
     FecMod  =  @L_FecMod  ,
     NumOrdProp  =  @P_NumOrdProp

   WHERE
     CodProp  =  @P_CodProp  AND
     CodPropComp  =  @P_CodPropComp

   SELECT @P_Error = @@ERROR , @P_RegModificado = @@ROWCOUNT

   IF @P_Error <> 0
      BEGIN
      SELECT  @msg  = 'ERROR de MODIFICACION en  TbCompProp .'
      GOTO Err_Act
      END

END /* Fin ELSE ExisteRegEnBase. */
/*                                                          */
/**  FIN DEL PROCEDIMIENTO.  **/
END

RETURN
/*                                                          */
/**  Tratamiento de ERRORES.  **/

Err_Act:

SELECT @P_Error = 1

PRINT @msg

RETURN

GO
```

Figura 6.34. Procedimiento almacenado sp_ActuTbCompProp.

```
/*  sp_ExisTbCompProp  */
/*---------------------*/

IF EXISTS (SELECT * FROM sysobjects WHERE
   id = object_id('sp_ExisTbCompProp') AND
   sysstat & 0xf = 4)
   DROP PROCEDURE sp_ExisTbCompProp
```

```
GO

CREATE PROCEDURE sp_ExisTbCompProp
/*                                                                  */
/**  REGLA DE EXISTENCIA: estructura_composicion_propiedad  **/
/*                                                                  */
/**  1.- Objetivo.  **/
/*                                                                  */
/**  - Comprobar existencia de elementos en estructura_composicion_propiedad  **/
/*                                                                  */
/**  2.- Declaracion de parametros del procedimiento.  **/

/*                                                                  */
/**  - Declaracion de identificadores de estructura_composicion_propiedad.  **/


@P_CodProp char (30) ,
@P_CodPropComp char (30) ,
/*                                                                  */
/**  - Declaracion de variable local 'error'.  **/

@P_Error int OUTPUT

AS
/*                                                                  */
/**  3.- Inicio del procedimiento.  **/

SELECT @P_Error = 0
/*                                                                  */
/**  - Si existe elemento en estructura_composicion_propiedad devuelve 1 (verdadero).  **/
/**  - Si no existe elemento en estructura_composicion_propiedad devuelve 0 (falso).  **/

IF EXISTS (

SELECT
  CodProp  ,
  CodPropComp

FROM TbCompProp

WHERE
  CodProp  =  @P_CodProp  AND
  CodPropComp  =  @P_CodPropComp )

RETURN (1)

ELSE

RETURN (0)
/*                                                                  */
/**  4.- Fin del procedimiento.  **/
/*                                                                  */
/**  - Si error, activar variable local 'error'.  **/

IF @@ERROR <> 0 SELECT @P_Error = 1

GO
```

Figura 6.35. Procedimiento almacenado sp_ExisTbCompProp.

La copia de los datos de cualquier otra tabla requeriría escribir procedimientos almacenados análogos a *sp_AprovTbCompProp, sp_ActuTbCompProp* y *sp_ExisTbCompProp*, lo que supone una cantidad de trabajo repetitivo enorme (362 + 164 + 66 ≈ 592 líneas de código por tabla). Es preferible tratar de generalizar los procedimientos *sp_AprovTbCompProp, sp_ActuTbCompProp* y *sp_ExisTbCompProp*. Sin embargo, la generalización deseada exige expresar de forma genérica el recorrido de las filas y de las

columnas de una tabla. TRANSACT SQL dispone de un mecanismo para expresar el recorrido secuencial de las filas, los **cursores**. Sin embargo, no tiene ningún medio para expresar de forma genérica el recorrido de las columnas de una tabla.

A continuación, se desarrollará con EDD y ETL una flexibilización de *sp_AprovTbCompProp, sp_ActuTbCompProp* y *sp_ExisTbCompProp*, que facilitará la obtención automática de procedimientos almacenados de este tipo.

6.3.1. Análisis de la familia de productos

La flexibilización producirá procedimientos almacenados sp_Aprov*NombreDeLaTabla,* sp_Actu*NombreDeLaTabla* y sp_Exis*NombreDeLaTabla*, que copiarán los datos de una tabla *NombreDeLaTabla* en otra temporal Tem*NombreDeLaTabla*.

La figura 6.36 es un diagrama FODA que modela la variabilidad del dominio.

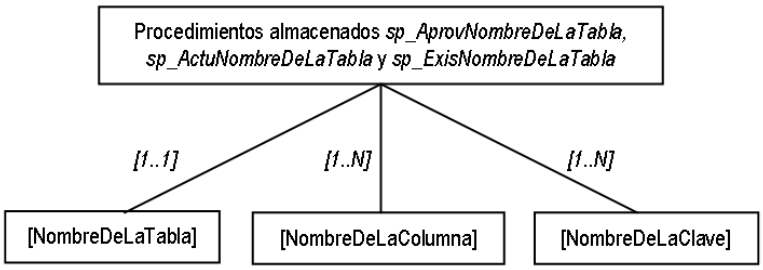


Figura 6.36. Modelo del dominio "copiadores de datos entre tablas".

6.3.2. Definición de una interfaz para la especificación abstracta de los productos

Para la generación de los procedimientos almacenados es imprescindible conocer el valor de los requisitos variables representados en la figura 6.36. Normalmente, para facilitar la especificación abstracta de estos valores se define un DSL. Sin embargo, en este ejemplo la especificación es innecesaria porque *Microsoft SQL Server* construye y actualiza automáticamente **metatablas**[65] con información sobre todas las tablas que gestiona. Concretamente, nuestro compilador extraerá el valor de los requisitos variables de las metatablas *syscolumns, sysobjects, sysindexkeys* y *systypes*.

La figura 6.37 es el del compilador de "copiadores de datos entre tablas".

[65] E. Arellano muestra en su tesis doctoral [Are98] el ahorro que supone el uso de metadatos en la especificación de sistemas de información.

La figura 6.38 incluye:

- el "análisis" que extrae, utilizando la librería DBI de Ruby [DuB06], la metainformación relativa a las tablas *TbAtrTb* y *TbTb*, mostradas en la figura 6.29

- la ejecución del generador *AprovTbGen*, que se presenta en la siguiente sección.

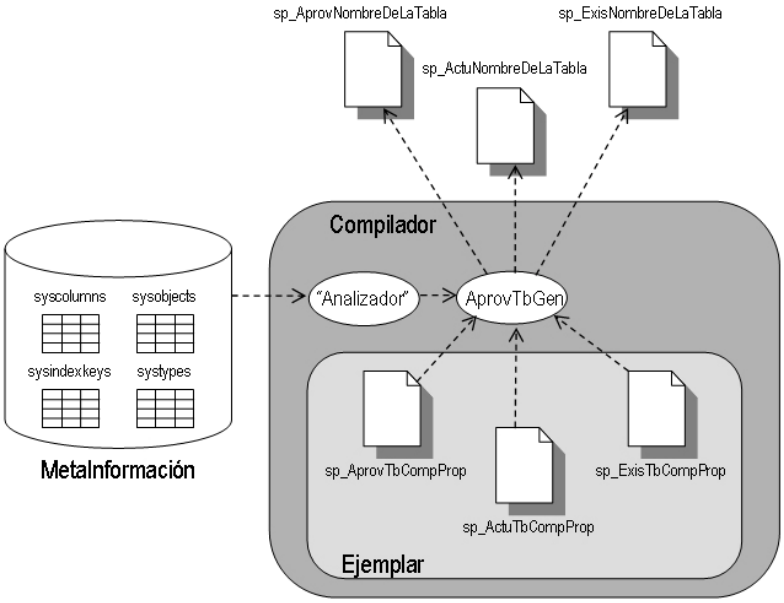


Figura 6.37. Diseño del compilador de "copiadores de datos entre tablas".

```
#######################################################
# Extracción de la especificación de la Base de Datos
#######################################################

require 'dbi'
dbh = DBI.connect("dbi:ODBC:BDatos")

tablas = ['TbAtrTb', 'TbTb']

tablas.each { |tabla|
  # Obtención de las columnas de la tabla
  sth = dbh.execute("
    SELECT syscolumns.name, systypes.name, syscolumns.length
      FROM sysobjects, syscolumns, systypes
      WHERE sysobjects.name = '#{tabla}'
        AND syscolumns.id = sysobjects.id
        AND syscolumns.xtype = systypes.xtype
  ")

  columnas = {}
  while row = sth.fetch do
    columnas[row[0]] = row[1] + "(#{row[2]})"
  end

  # Obtención de las claves de la tabla
  sth = dbh.execute("
  SELECT DISTINCT syscolumns.name
    FROM syscolumns, sysobjects, sysindexkeys
    WHERE sysobjects.name = '#{tabla}'
      AND syscolumns.colid = sysindexkeys.colid
      AND syscolumns.id = sysobjects.id
      AND sysindexkeys.id = sysobjects.id
  ")
```

```
  claves = []
  while row = sth.fetch do
    claves += [row[0]]
  end

  ########################################
  # Ejecución del generador
  ########################################

  AprovTbGen.new(tabla, columnas, claves).gen
}
```

Figura 6.38. "Análisis" de la especificación de los procedimientos almacenados para las tablas TbAtrTb y TbTb. Ejecución del generador AprovTbGen.

6.3.3. Implementación de la flexibilización del ejemplar

La figura 6.39 muestra la actuación del generador *AprovTvGen* para obtener los procedimientos almacenados de la tabla *TbAtrTb*, incluida en la figura 6.29. La figura 6.40 es el código del generador. Se han utilizado los colores ○, ○, ○, ○, ○, ○, ○, ○, ○, ○, ○, ○, ○ y ○ para identificar el código variable en el ejemplar (figuras 6.32, 6.34 y 6.35) y las sustituciones del generador que lo adaptan (figuras 6.39 y 6.40).

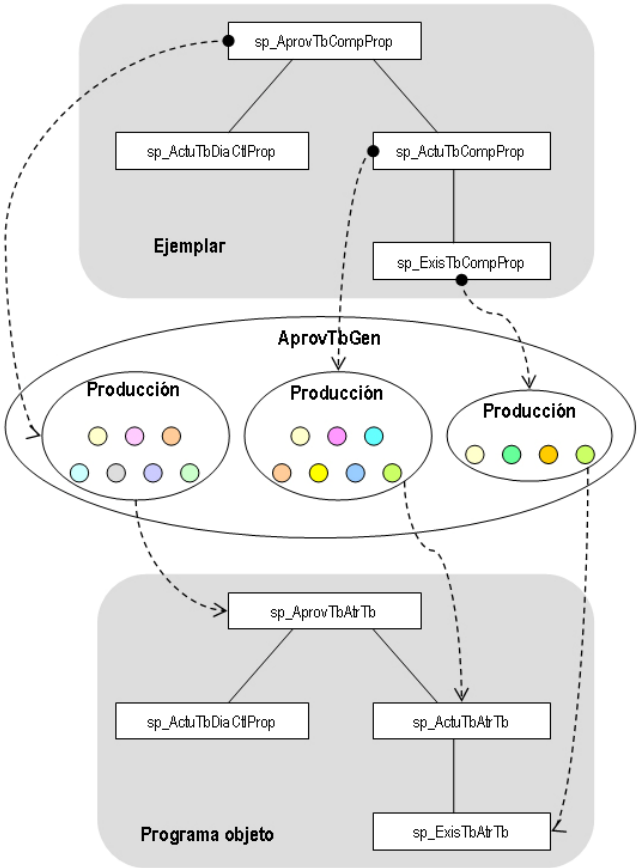


Figura 6.39. Ejemplo de generación de los procedimientos almacenados que copian los datos de la tabla TbAtrTb.

```
class AprovTbGen < Generator
  def initialize(nombreTabla, columnas, claves)

    ##############################################################
    # Eliminación del ancho en las columnas de tipo: int, text
    ##############################################################

    columnas.each {|columna, tipo|
      if tipo =~ /(int|text)/
        columnas[columna] = $1
      end
    }

    ##########################################
    # Sustituciones
    ##########################################

    gsub(/TbCompProp/, nombreTabla,
      'nombre de la tabla')

    sub(/((DECLARE @L_).+$\n)+/,
      '\2' + columnas.keys + ' ' + columnas.values,
      'registros locales para tratar la tabla base'
    )

    gsub(/\bCodProp\s*,\s*\n(.*,\s*)+NumOrdProp/,
      columnas.keys.join(", \n  "),
      'lista de las columnas de la tabla base unidas por comas'
    )

    sub(/(ORDER BY\s*).*,\n.*/,
      '\1' + claves.join(", \n\t"),
      'ORDER BY'
    )

    gsub(
      /
      @L_CodProp\s*,\s*
      (@L_\w+\s*,\s*)+
      @L_NumOrdProp\s*?,?\s*?
      /x,
      ('@L_' + columnas.keys ).join(",\n"),
      'lista de las columnas de la tabla base precedidas por @L_ unidas por comas'
    )

    sub(
       /
      @L_CodProp\s*,\s*
      (@L_\w+\s*,\s*)+
      (@P_CodUsu\s*,\s*)
      (@L_\w+\s*,\s*)+
      @L_NumOrdProp
      /x,
      columnas.keys.collect {|c|
        if c == 'CodUsu'
          "@P_#{c}"
        else
          "@L_#{c}"
        end
      }.join(",\n"),
      'ejecución de sp_ActuTbCompProp'
    )

    sub(/@L_CodProp \s*\+\s*''\s*\+\s*@L_CodPropComp/,
      ('@L_' + claves).join(" + '' + "),
      'SELECT @L_UltClaveActu'
    )
```

```
texto = ''
columnas.each {|columna, tipo|
    if claves.include? columna
      texto += "@P_#{columna} #{tipo}, --Identificador de la estructura\n"
    else
      texto +="@P_#{columna} #{tipo}, --Descriptor de la estructura\n"
    end
  }
sub(
  /
  \/\*\* \s*Identificadores\s*de\s*la\s*estructura
  .*
  (\n\/\*\* \s*Otros\s*parametros\s*del\s*procedimiento)
  /mx,
  texto + '\1',
  'declaracion de parametros del procedimiento sp_ActuTb'
  )
```

```
sub(
  /
  @P_CodProp\s*,\s*
  @P_CodPropComp\s*,
  (?=\s*@L_ErrorExistencia)
  /x,
  '@P_' + claves + ",",
  'ejecución de sp_ExisteTbCompProp'
)
```

```
sub(
   /
  @P_CodProp\s*,\s*
  (@[PL]_\w+\s*,\s*)+
  @P_NumOrdProp
  /x,
  columnas.keys.collect {|c|
    if ['FecAlta', 'FecMod', 'FecBaja'].include? c
      "@L_#{c}"
    else
      "@P_#{c}"
    end
  }.join(",\n"),
  'VALUES de INSERT INTO TbCompProp'
)
```

```
lista = columnas.keys - claves - ['FecAlta', 'FecBaja']
sub(/
  CodUsu\s*=.*\s*
  FecMod\s*=.*\s*
  NumOrdProp\s*=.*
  /x,
  lista.collect {|c|
    if c =~ /Fec/
      "#{c} = @L_#{c}"
    else
      "#{c} = @P_#{c}"
    end
  }.join(",\n"),
  'UPDATE TbCompProp'
)
```

```
sub(/
  CodProp\s*=\s*@P_CodProp\s*AND\s*
  CodPropComp\s*=\s*@P_CodPropComp
  /x,
  claves.collect {|c|
    "#{c} = @P_#{c}"
  }.join(" AND\n"),
  'WHERE claves'
)
```

```
texto = ''
claves.each { |c|
  texto += "@P_#{c} #{columnas[c]} ,\n"
```

```
    }
    sub(/
      @P_CodProp\s*char\s*\(30\)\s*,\s*
      @P_CodPropComp\s*char\s*\(30\)\s*,
         /x,
         texto,
         'declaracion de parametros del procedimiento'
    )

    sub(/CodProp\s*,\s*CodPropComp/,
      claves.join(",\n"),
      'SELECT claves'
    )


    ##########################################
    # Producciones
    ##########################################

    prod('Ejemplares/sp_AprovTbCompProp.sql',
      "out/sp_Aprov#{nombreTabla}.sql",
      ['nombre de la tabla',
      'registros locales para tratar la tabla base',
      'lista de las columnas de la tabla base unidas por comas',
      'ORDER BY',
      'lista de las columnas de la tabla base precedidas por @L_ unidas por comas',
      'ejecución de sp_ActuTbCompProp',
      'SELECT @L_UltClaveActu'
      ]
    )

    prod('Ejemplares/sp_ActuTbCompProp.sql',
      "out/sp_Actu#{nombreTabla}.sql",
      ['nombre de la tabla',
      'declaracion de parametros del procedimiento sp_ActuTb',
      'ejecución de sp_ExisteTbCompProp',
      'lista de las columnas de la tabla base unidas por comas',
      'VALUES de INSERT INTO TbCompProp',
      'UPDATE TbCompProp',
      'WHERE claves'
      ]
    )

    prod('Ejemplares/sp_ExisTbCompProp.sql',
      "out/sp_ExisTb#{nombreTabla}.sql",
      ['nombre de la tabla',
      'declaracion de parametros del procedimiento',
      'SELECT claves',
      'WHERE claves'
      ]
    )
  end

end
```

Figura 6.40. Generador AprovTbGen.

6.4. Prueba de unidades

"Las pruebas son una fase cara y laboriosa del proceso de software" [Som05, página 513]. Sin embargo, gracias a la aparición de herramientas como *JUnit* para Java [JUn06], *NUnit* para C# [NUn06], *CppUnit* para C++ [Cpp06], *RubyUnit* para Ruby [RUn06], *HttpUnit* para aplicaciones web [HUn06]… es posible automatizar las pruebas de unidades, reduciéndose así los costes de la fase de pruebas.

En esta sección se utiliza ETL para desarrollar la herramienta **m2unit** para las prueba de unidades de programas escritos en **Modula-2**.

6.4.1. Manual de usuario de m2unit

m2unit tiene las siguientes características:

- Como indica la *eXtreme Programming* [Bec02], la realización de pruebas de unidades debería formar parte de la rutina diaria del programador. Es decir, el principal destinatario de m2unit es el programador de Modula-2. Por eso, la notación de m2unit es una pequeña extensión de Modula-2 que únicamente añade la palabra clave **`TEST`**, para la definición de un juego de pruebas, y el procedimiento **`Assert`**, para la verificación de expresiones booleanas.

- Para sensibilizar al programador sobre la importancia de realizar pruebas, éstas se embeben en el código.

- Para permitir que los programas enriquecidos con pruebas sean traducibles a código máquina por cualquier compilador de Modula-2, las pruebas se introducen como comentarios (entre los símbolos `(*` y `*)`).

Por ejemplo, supongamos que se desea probar el módulo de implementación `Util` de la figura 6.41. Para ello, habría que incrustarle las pruebas `Test1` y `Test2` como muestra la figura 6.42.

```
IMPLEMENTATION MODULE Util;

PROCEDURE Exp(base, exponente: INTEGER): INTEGER;
VAR
  i: INTEGER;
BEGIN
  IF exponente = 0 THEN
    RETURN 1;
  ELSE
    RETURN base * Exp(base, exponente - 1);
  END; (* IF *)
END Exp;

PROCEDURE IntercambiarValores(VAR x,y: INTEGER);
VAR
  z: INTEGER;
BEGIN
  z := x;
  x := y;
  y := z;
END IntercambiarValores;

END Util.
```

Figura 6.41.Módulo Util.

```
IMPLEMENTATION MODULE Util;

PROCEDURE Exp(base, exponente: INTEGER): INTEGER;
VAR
  i: INTEGER;
BEGIN
  IF exponente = 0 THEN
    RETURN 1;
  ELSE
    RETURN base * Exp(base, exponente - 1);
  END; (* IF *)
END Exp;

(*
  TEST Test1;
  VAR
    x: INTEGER;
  BEGIN

    Assert(Exp(2,2) = 4);
    x := 8;
    Assert(x = Exp(2,3));
  END Test1;
*)

PROCEDURE IntercambiarValores(VAR x,y: INTEGER);
VAR
  z: INTEGER;
BEGIN
  z := x;
  x := y;
  y := z;
END IntercambiarValores;

(*
  TEST Test2;
  VAR
    x, y: INTEGER;
  BEGIN
     x := 1;
     y := 2;
     IntercambiarValores(x, y);
    Assert(x = 2);
    Assert(y = 1);
  END Test2;
*)

END Util.
```

Figura 6.42. Módulo Util enriquecido con las pruebas Test1 y Test2.

A continuación, se ejecutaría m2unit escribiendo en la línea de comandos: `m2unit Util.mod`[66]. La figura 6.43 sería el resultado de la ejecución.

```
     m2unit alpha version. (2-12-2007)
        Unit Test for Modula-2 code
-----------------------------------------------
 author: Ruben Heradio (rheradio@issi.uned.es)
```

[66] La notación de m2unit es `m2unit módulo_1 [módulo_2] … [módulo_n]`

```
=============================================
Running Test1 from module Util
---------------------------------------------
---------------------------------------------
Test OK
=============================================


=============================================
Running Test2 from module Util
---------------------------------------------
---------------------------------------------
Test OK
=============================================
```

Figura 6.43. Resultado de ejecutar con m2unit la figura 6.42.

6.4.2. Implementación de m2unit

m2unit recorre cada módulo que recibe como argumento en busca de tests. Cuando localiza un test, transforma el módulo que lo contiene en un módulo principal que ejecuta el test[67]. A continuación, compila el módulo principal obtenido con el compilador FST de Modula-2 y ejecuta el código objeto. Finalmente, tras la ejecución de los tests, elimina los ficheros generados.

m2unit transforma los módulos aplicando secuencialmente los generadores *MainProgramGenerator, DeleteTestCommentaries, ImportGenerator, AssertGenerator* y *TypeAndConstInclusion*. La figura 6.44 muestra la actuación de los generadores sobre el módulo de la figura 6.42 para obtener los módulos principales de las figuras 6.45 y 6.46.

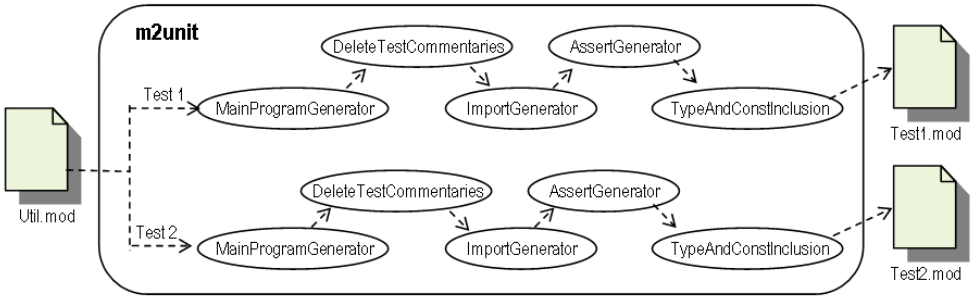


Figura 6.44. Ejemplo de actuación de los generadores de m2unit.

```
MODULE Test1;

FROM InOut IMPORT
  OpenInput, OpenOutput, RedirectInput, RedirectOutput, CloseInput, CloseOutput,
  Read, ReadString, ReadLine, ReadInt, ReadCard, ReadWrd,
  Write, WriteLn, WriteString, WriteLine, WriteInt, WriteCard, WriteOct, WriteHex, WriteWrd,
  ReadLongInt, ReadLongCard, WriteLongInt, WriteLongCard;
```

[67] Cada módulo de entrada actúa simultáneamente como programa fuente y como ejemplar.

```
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

PROCEDURE Exp(base, exponente: INTEGER): INTEGER;
VAR
  i: INTEGER;
BEGIN
  IF exponente = 0 THEN
    RETURN 1;
  ELSE
    RETURN base * Exp(base, exponente - 1);
  END; (* IF *)
END Exp;

PROCEDURE IntercambiarValores(VAR x,y: INTEGER);
VAR
  z: INTEGER;
BEGIN
  z := x;
  x := y;
  y := z;
END IntercambiarValores;

VAR
  x: INTEGER;
VAR
  TestOK: BOOLEAN;      (* TRUE if the test is free of failures *)
  assertNum: CARDINAL;  (* assert's counter *)

BEGIN
  TestOK := TRUE;
  assertNum := 0;
  WriteLn;
  WriteString("==============================================");
  WriteLn;
  WriteString("Running Test1 from module Util");
  WriteLn;
  WriteString("----------------------------------------------");
  WriteLn;

  INC(assertNum);
  IF NOT((Exp(2,2) = 4)) THEN
    TestOK := FALSE;
    WriteString("   - assert number ");
    WriteCard(assertNum,0);
    WriteString(" failed");
    WriteLn;
  END;

  x := 8;
  INC(assertNum);
  IF NOT((x = Exp(2,3))) THEN
    TestOK := FALSE;
    WriteString("   - assert number ");
    WriteCard(assertNum,0);
    WriteString(" failed");
    WriteLn;
  END;

  IF TestOK THEN
    WriteString("----------------------------------------------");
    WriteLn;
    WriteString("Test OK");
    WriteLn;
    WriteString("==============================================");
    WriteLn;
  ELSE
    WriteString("----------------------------------------------");
    WriteLn;
    WriteString("Test failed");
    WriteLn;
    WriteString("==============================================");
    WriteLn;
  END;
```

```
END Test1.
```

Figura 6.45. Módulo principal Test1 generado por m2unit.

```
MODULE Test2;

FROM InOut IMPORT
     OpenInput, OpenOutput, RedirectInput, RedirectOutput, CloseInput, CloseOutput,
     Read, ReadString, ReadLine, ReadInt, ReadCard, ReadWrd,
     Write,  WriteLn,  WriteString,  WriteLine,  WriteInt,  WriteCard,  WriteOct,  WriteHex,
WriteWrd,
     ReadLongInt, ReadLongCard, WriteLongInt, WriteLongCard;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;

PROCEDURE Exp(base, exponente: INTEGER): INTEGER;
VAR
  i: INTEGER;
BEGIN
  IF exponente = 0 THEN
    RETURN 1;
  ELSE
    RETURN base * Exp(base, exponente - 1);
  END; (* IF *)
END Exp;

PROCEDURE IntercambiarValores(VAR x,y: INTEGER);
VAR
  z: INTEGER;
BEGIN
  z := x;
  x := y;
  y := z;
END IntercambiarValores;

VAR
  x, y: INTEGER;
VAR
  TestOK: BOOLEAN;      (* TRUE if the test is free of failures *)
  assertNum: CARDINAL;  (* assert's counter *)

BEGIN
  TestOK := TRUE;
  assertNum := 0;
  WriteLn;
  WriteString("==============================================");
  WriteLn;
  WriteString("Running Test2 from module Util");
  WriteLn;
  WriteString("----------------------------------------------");
  WriteLn;

  x := 1;
  y := 2;
  IntercambiarValores(x, y);
  INC(assertNum);
  IF NOT((x = 2)) THEN
    TestOK := FALSE;
    WriteString("   - assert number ");
    WriteCard(assertNum,0);
    WriteString(" failed");
    WriteLn;
  END;

  INC(assertNum);
  IF NOT((y = 1)) THEN
    TestOK := FALSE;
    WriteString("   - assert number ");
    WriteCard(assertNum,0);
    WriteString(" failed");
    WriteLn;
```

```
  END;

  IF TestOK THEN
    WriteString("----------------------------------------------");
    WriteLn;
    WriteString("Test OK");
    WriteLn;
    WriteString("==============================================");
    WriteLn;
  ELSE
    WriteString("----------------------------------------------");
    WriteLn;
    WriteString("Test failed");
    WriteLn;
    WriteString("==============================================");
    WriteLn;
  END;

END Test2.
```

Figura 6.46. Módulo principal Test2 generado por m2unit.

Finalmente, la figura 6.47 es el código de m2unit.

```
require '..\..\ETL\ETL'

class MainProgramGenerator < Generator
  def initialize(fileIn, dirOut,  testName)
    @fileIn = fileIn
    @testName = testName
    @sourceModuleName = extract(fileIn, /MODULE\s+(\w+)\s*;/m)[1]
    @testCode = extract(fileIn, /TEST\s+(#{@testName})\s*;.+?END\s+\1\s*;/m)

    if @testCode == nil
      raise Exception,
        "on module #{@fileIn}, the test #{@testName} " +
        "is empty or has some \nsyntactic errors"
    end

    moduleNameCode()
    varCode()
    mainCode()
    prod(fileIn, "#{dirOut}\\#{testName}.mod")
  end

  def moduleNameCode()
    sub(/((IMPLEMENTATION|DEFINITION)\s+)?(MODULE\s+)\w+(\s*;)/m,
      '\3' + @testName + '\4')
    sub(/(END\s+)\w+(\s*\.)/, '\1' + @testName + '\2')
  end

  def varCode()
    localTestVarCodeRegExp = /
      (\s*VAR\s+
      .+?)
      BEGIN
    /xm

    if @testCode[0] =~ localTestVarCodeRegExp
      varCode = $1
    else
      varCode = ''
    end

    varCode += <<-END_OF_MODULA2_CODE
      VAR
        TestOK: BOOLEAN;      (* TRUE if the test is free of failures *)
        assertNum: CARDINAL;  (* assert's counter *)
    END_OF_MODULA2_CODE
```

```
    beginRegExp = /
      (?=
        (
        BEGIN
          ((?!(END\s+\w+;)).)+?
        END\s+\w+\.
        )
      )
    /xm
    if extract(@fileIn, beginRegExp)
      sub(beginRegExp, varCode)
    else
      sub(/(?=(END\s+\w+\s*\.))/m, varCode)
    end
  end

  def mainCode()
    insertBeginCode()

    beginCode = <<-END_OF_MODULA2_CODE
      TestOK := TRUE;
      assertNum := 0;
      WriteLn;
      WriteString("==============================================");
      WriteLn;
      WriteString("Running #{@testName} from module #{@sourceModuleName}");
      WriteLn;
      WriteString("----------------------------------------------");
      WriteLn;
    END_OF_MODULA2_CODE

    programCode = @testCode.to_s.scan(/BEGIN(.+?)END\s+#{@testName}\s*;/m).to_s

    endCode = <<-END_OF_MODULA2_CODE
      IF TestOK THEN
        WriteString("----------------------------------------------");
        WriteLn;
        WriteString("Test OK");
        WriteLn;
        WriteString("==============================================");
        WriteLn;
      ELSE
        WriteString("----------------------------------------------");
        WriteLn;
        WriteString("Test failed");
        WriteLn;
        WriteString("==============================================");
        WriteLn;
      END;
    END_OF_MODULA2_CODE

    code = beginCode + programCode + endCode

    sub(/(?=(END\s+\w+\s*\.))/, code)
  end

  def insertBeginCode()
    beginRegExp = /
      BEGIN
        ((?!(END\s+\w+;)).)+?
      END\s+\w+\.
    /xm
    if !extract(@fileIn, beginRegExp)
      sub(/(?=(END\s+\w+\s*\.))/m, "BEGIN\n")
    end
  end

end

class DeleteTestCommentaries < Generator
  def initialize(fileIn, fileOut = fileIn)
    testCommentary = /
      TEST\s+(\w+)\s*;
```

```
      .+?
      END\s+\1\s*;
    /mx
    emptyCommentary = /\(\*\s*?\*\)/
    gsub(testCommentary, '')
    gsub(emptyCommentary, "\n")
    prod(fileIn, fileOut)
  end
end

class ImportGenerator < Generator
  def initialize(fileIn, fileOut = fileIn)
    @fileIn = fileIn
    @fileOut = fileOut
    @importCode = "\n"

    importSub(method(:inOutTemplate), 'InOut')
    importSub(method(:storageTemplate), 'Storage')
    sub(/(MODULE.+?;)/m, '\1'+@importCode)
    prod(fileIn, fileOut)
  end

  def importSub(modula2Template, moduleName)
    if extract(@fileIn, /FROM\s+#{moduleName}/)
      sub(/FROM\s+#{moduleName}.+?;/m, modula2Template.call)
    elsif extract(@fileIn, /IMPORT\s+#{moduleName}/)
    else
      @importCode += modula2Template.call
    end
  end

  def inOutTemplate()
    code = <<-END_OF_MODULA2_CODE
      FROM InOut IMPORT
        OpenInput, OpenOutput, RedirectInput, RedirectOutput, CloseInput, CloseOutput,
        Read, ReadString, ReadLine, ReadInt, ReadCard, ReadWrd,
        Write, WriteLn, WriteString, WriteLine, WriteInt, WriteCard, WriteOct, WriteHex,
        WriteWrd, ReadLongInt, ReadLongCard, WriteLongInt, WriteLongCard;
    END_OF_MODULA2_CODE
  end

  def storageTemplate()
    code = <<-END_OF_MODULA2_CODE
      FROM Storage IMPORT ALLOCATE, DEALLOCATE;
    END_OF_MODULA2_CODE
  end

end

class AssertGenerator < Generator
  def initialize(fileIn, fileOut = fileIn)
    gsub(/Assert(.+?);/, delay('#{template($1)}'))
    prod(fileIn, fileOut)
  end

  def template(condition)
    code = <<-END_OF_MODULA2_CODE
      INC(assertNum);
      IF NOT(#{condition}) THEN
        TestOK := FALSE;
        WriteString("   - assert number ");
        WriteCard(assertNum,0);
        WriteString(" failed");
        WriteLn;
      END;
    END_OF_MODULA2_CODE
  end
end

class TypeAndConstInclusion < Generator
  def initialize(defFile, fileIn, fileOut = fileIn)
    constRegExp = /CONST\s+((?!CONST)(?!TYPE).+?(?!:)=.+?;\s*)+/m
    constMatch = extract(defFile, constRegExp)
```

```
    typeRegExp = /TYPE\s+((?!CONST)(?!TYPE).+?(?!:)=.+?;\s*)+/m
    typeMatch = extract(defFile, typeRegExp)

    code = ''
    code += constMatch[0] if constMatch
    code += typeMatch[0] if typeMatch

    importRegExp = /((FROM|IMPORT).+?;\s*)+/m
    sub(importRegExp, delay('#{$&}' + code))

    prod(fileIn, fileOut)
  end
end


# Main Program
AUTHOR_INFO = <<-END_AUTHOR_INFO

     m2unit alpha version. (2-12-2007)
        Unit Test for Modula-2 code
-----------------------------------------------
 author: Ruben Heradio (rheradio@issi.uned.es)

END_AUTHOR_INFO


TEST_DIR = 'Test'

unless ARGV[0]
  print "m2unit usage: m2unit file_1 [file_2]...[file_n]\n"
  exit
end

puts AUTHOR_INFO


Dir.mkdir(TEST_DIR) unless FileTest.exist?(TEST_DIR)
ARGV.each {|arg|
   begin
  file_dir = "#{TEST_DIR}\\#{arg}".scan(/(.+?)\.\w+/i).to_s
  File.open(arg) {|file|
    arg =~ /(.+?)\.mod$/
    defFile = "#{$1}.def"
    code = file.read
    Dir.mkdir(file_dir) unless FileTest.exist?(file_dir)
    tests = code.scan(/TEST\s+(\w+);/).flatten
    tests.each {|test|
      MainProgramGenerator.new(arg, file_dir, test).gen
      DeleteTestCommentaries.new("#{file_dir}\\#{test}.mod").gen
      ImportGenerator.new("#{file_dir}\\#{test}.mod").gen
      AssertGenerator.new("#{file_dir}\\#{test}.mod").gen
      if FileTest.exist?(defFile)
        TypeAndConstInclusion.new(defFile, "#{file_dir}\\#{test}.mod").gen
      end
      if (`m2comp #{file_dir}\\#{test}` =~ /error/mi) or
        !(`m2comp #{file_dir}\\#{test}` =~ /Pass\s+2/i)
        raise Exception,
          "on module #{arg}, the test #{test} can't be compiled. " +
          "\nPlease, check your code..."
        exit
      end
      if `m2link #{file_dir}\\#{test} /o` =~ /error/mi
        raise Exception,
          "on module #{arg}, the test #{test} can't be linked. "
          "\nPlease, check your code..."
      end
      puts `#{file_dir}\\#{test}` + "\n"
    }
  }
   rescue Exception => message
  puts "<< ERROR: #{message} >>"
   end
}
```

```
`rmdir /s /q #{TEST_DIR}`
```

Figura 6.47. Código de m2unit.

6.5. Aumento de la expresividad y la concisión de un lenguaje

ETL puede utilizarse para mejorar la expresividad y la concisión de un lenguaje. Por ejemplo, **Java** dispone del tipo básico *double* para el cálculo numérico de alta precisión. Sin embargo, para evitar desbordamientos en operaciones con números muy grandes, Java ofrece la clase *BigDecimal*.

Lamentablemente, la escritura de operaciones con *BigDecimal* es tediosa porque:

- Impone el uso de *castings* (*double* → *BigDecimal*) y de variables temporales.
- Java no soporta la sobrecarga de operadores, por lo que frente a la notación aritmética convencional habrá que utilizar una notación funcional.

Por ejemplo, la ecuación `a=((b+c)*d)/e` de la figura 6.48 deberá escribirse como indica la figura 6.49.

```
import java.math.*;

public class Test
{
  static public void main( String args[] )
  {
    BigDecimal a;
    double b = 4.0;
    double c = 6.0;
    double d = 1.0;
    double e = 2.0;

    // BigDecimalEquation: a = ((b + c) * d)/e

    System.out.println( "Correct = 5" );
    System.out.print( "Output = " );
    System.out.println( a );
  }
}
```

Figura 6.48. Programa Java enriquecido con la notación aritmética para la clase BigDecimal.

```
import java.math.*;

public class Test
{
  static public void main( String args[] )
  {
    BigDecimal a;
    double b = 4.0;
    double c = 6.0;
```

```
    double d = 1.0;
    double e = 2.0;

    // BigDecimalEquation: a = ((b + c) * d)/e
       BigDecimal tmp_3 = new BigDecimal(b);
       BigDecimal tmp_5 = new BigDecimal(c);
       BigDecimal tmp_4 = new BigDecimal(0);
       tmp_4 = tmp_3.add(tmp_5);
       BigDecimal tmp_8 = new BigDecimal(d);
       BigDecimal tmp_7 = new BigDecimal(0);
       tmp_7 = tmp_4.multiply(tmp_8);
       BigDecimal tmp_11 = new BigDecimal(e);
       BigDecimal tmp_10 = new BigDecimal(0);
       tmp_10 = tmp_7.divide(tmp_11, BigDecimal.ROUND_DOWN);
       a = tmp_10;
    // EndBigDecimalEquation

    System.out.println( "Correct = 5" );
    System.out.print( "Output = " );
    System.out.println( a );
  }
}
```

Figura 6.49. Programa Java convencional equivalente a la figura 6.48.

A continuación, se construirá con ETL un prepocesador que extienderá Java para soportar la notación aritmética de la figura 6.48 y producirá programas del estilo de la figura 6.49.

6.5.1. Interfaz de la extensión

Como puede apreciarse en la figura 6.48, las ecuaciones serán comentarios con la notación:

`// BigDecimalEquation: Ecuación`

6.5.2. Preprocesador ETL

Los programas java enriquecidos servirán como programas fuente y como ejemplares del preprocesador. El **generador** *BigDecEqGen* de la figura 6.50 se limitará a expresar la sustitución de la ecuación en notación aritmética por su equivalente en Java convencional. Finalmente, para el **análisis** de la ecuación y la obtención del equivalente Java se utilizará un compilador auxiliar "clásico", escrito aprovechando el meta-analizador **Racc** [Racc05] (figura 6.51).

```
require '..\..\ETL\ETL'
require 'bigdeceqparser'

class BigDecEqGen < Generator
        def initialize(fin, fout)
                bDRegExp = /(\s*\/\/\s*BigDecimalEquation\s*:\s*)
                                (.+?$)
                                ( (.*?)
                                  (\s*\/\/\s*EndBigDecimalEquation\s*$) )?
                                /xm
                @parser = BigDecEqParser.new(:Lexer)
                @endCode = "// EndBigDecimalEquation\n"
                gsub(bDRegExp, delay('#{$1+$2}\n#{@parser.parse($2)}\n#{@endCode}'))
```

```
                prod(fin, fout)
        end
end

unless ARGV[0] && ARGV[1]
        print "bdgen usage: bdgen file_in file_out\n"
        exit
end

BigDecEqGen.new(ARGV[0], ARGV[1]).gen
```

Figura 6.50. Preprocesador BigDecEqGen escrito en ETL.

```
class BigDecEqParser

  prechigh
    left '*' '/'
    left '+' '-'
  preclow

rule

  target: ID '=' exp {
        result = val[2].code + "#{val[0].value} = tmp_#{val[2].tmp};\n"
        }
        | /* none */ { result = '' }
        ;

  exp: exp '+' exp {
          result = TokenValueClass.new(
                       val[1].value,
                       val[1].tmp,
                       val[0].code + val[2].code +
                       "BigDecimal tmp_#{val[1].tmp} = new BigDecimal(0);\n" +
                       "tmp_#{val[1].tmp} = tmp_#{val[0].tmp}.add(tmp_#{val[2].tmp});\n"
                     )
        }
     | exp '-' exp  {
          result = TokenValueClass.new(
                       val[1].value,
                       val[1].tmp,
                       val[0].code + val[2].code +
                       "BigDecimal tmp_#{val[1].tmp} = new BigDecimal(0);\n" +
                       "tmp_#{val[1].tmp} = tmp_#{val[0].tmp}.subtract(tmp_#{val[2].tmp});\n"
                     )
           }
     | exp '*' exp {
          result = TokenValueClass.new(
                       val[1].value,
                       val[1].tmp,
                       val[0].code + val[2].code +
                       "BigDecimal tmp_#{val[1].tmp} = new BigDecimal(0);\n" +
                       "tmp_#{val[1].tmp} = tmp_#{val[0].tmp}.multiply(tmp_#{val[2].tmp});\n"
                     )
          }
     | exp '/' exp {
          result = TokenValueClass.new(
                       val[1].value,
                       val[1].tmp,
                       val[0].code + val[2].code +
                       "BigDecimal tmp_#{val[1].tmp} = new BigDecimal(0);\n" +
                       "tmp_#{val[1].tmp} = tmp_#{val[0].tmp}.divide(tmp_#{val[2].tmp},
                         BigDecimal.ROUND_DOWN);\n"
                     )
          }
     | '(' exp ')' {
           result = TokenValueClass.new(
                       val[1].value,
                       val[1].tmp,
                       val[1].code)
          }
```

```
     | ID '(' exp ')' {
           result = TokenValueClass.new(
                       val[1].value,
                       val[1].tmp,
                       val[2].code +
                       "BigDecimal tmp_#{val[1].tmp} =
                           #{val[0].value}(tmp_#{val[2].tmp});\n")
          }
     | ID {
           result = TokenValueClass.new(
                       val[0].value,
                       val[0].tmp,
                        "BigDecimal tmp_#{val[0].tmp} = new BigDecimal(#{val[0].value});\n"
                     )
           }
     ;

end

---- inner ----

def initialize(lexerClass)
  @lexer_class = lexerClass
end

def parse(aString)
  @lexer = eval "#{@lexer_class}.new(aString)"
  do_parse
end

def next_token
  @lexer.next_token
end

TokenValueClass = Struct.new("TokenValueClass", :value, :tmp, :code)

class LexerError < Exception; end

class Lexer
  @@tmp = 0

  def initialize(aString)
    @str = aString
  end

  def next_token
    token = nil
    until token
      if @str =~ /^[a-zA-Z_]([a-zA-Z_]|\d)*(\.[a-zA-Z_]([a-zA-Z_]|\d)*)*/
        token_value = TokenValueClass.new($&, @@tmp, '')
        token = [:ID, token_value]
        @@tmp +=1
      elsif @str =~ /^\s+/
        # do nothing
      elsif @str =~ /^(\+|-|\(|\)|\/|\*)/
        token_value = TokenValueClass.new($&, @@tmp, '')
        token = [$&, token_value]
        @@tmp +=1
      elsif  @str =~ /^=/
        token_value = TokenValueClass.new($&, @@tmp, '')
        token = [$&, token_value]
      elsif @str.length > 0
        raise LexerError, "Invalid token at '#{@str[0,10]}'"
      else
        token = [false, '$end']
      end # if
      @str = $'
    end # until
    return token
  end
end
```

Figura 6.51. Compilador BigDecEqParser escrito en Racc.

7

Conclusiones y trabajo futuro

In every project a large portion of the work done by developers is not genuinely creative - it is tedious routine work that we should try to automate [...] We may be able to get from 20/80 (20% creative content and 80% noncreative) to 80/20; I don't think we can eliminate it entirely.

I. Jacobson. *A resounding Yes to agile processes -- but also to more.*

7.1. Conclusiones

En el capítulo 1 se ha señalado la importancia de desarrollar familias de productos en lugar de productos aislados para conseguir una reutilización sistemática del software y lograr economía de alcance.

Los procesos de desarrollo de familias de productos se suelen descomponer en dos grandes actividades: la realización de una infraestructura que implemente de manera global los requisitos de todos los productos de una familia y la obtención posterior de cada producto, parametrizando dicha infraestructura. En general, se persigue facilitar la parametrización construyendo infraestructuras con ocultación de "caja negra", que dispongan de DSLs para la especificación abstracta de los productos y de compiladores para la generación automática de los productos finales a partir de las especificaciones DSL.

En el capítulo 2 se han resumido:

- los principales logros y áreas de investigación en la reutilización de productos software.

- recientes e importantes procesos de desarrollo de familias de productos.

- distintas estrategias para construir compiladores de DSLs.

En el capítulo 3 se ha presentado el proceso EDD de desarrollo de familias de productos. EDD es una aportación completamente original de esta tesis y propone aprovechar la similitud entre los productos de una familia para construirlos por analogía. La primera actividad de EDD es la realización de un producto concreto de una familia. A continuación, se busca cómo flexibilizar este ejemplar para que satisfaga los requisitos del resto de los productos. Es decir, se trata de definir una relación de analogía que permita derivar los demás productos del ejemplar. Por último, se obtienen todos los productos parametrizando la flexibilización del ejemplar mediante especificaciones DSL.

En el capítulo 4 se ha resumido el lenguaje ETL y su implementación en Ruby. ETL es una aportación completamente original de esta tesis y permite flexibilizar ejemplares escritos en cualquier lenguaje (código ejecutable, juegos de prueba, documentación en lenguaje natural…), superando las limitaciones de las técnicas de generalización de código examinadas en el capítulo 5.

En capítulo 5 se han estudiado distintas maneras de flexibilizar un ejemplar mediante las técnicas más comunes de generalización de código (herencia, genericidad, aspectos, plantillas de código…). Lamentablemente, se ha comprobado que estas técnicas son insuficientes para conseguir flexibilizaciones satisfactorias (modulares, no invasivas, aplicables a cualquier producto software…), lo que justifica la necesidad de ETL. Además, en este capítulo se ha utilizado un supuesto práctico para resaltar las ventajas de EDD frente a otros procesos de desarrollo. Algunas de estas ventajas son:

- Abordar el desarrollo y el mantenimiento de una familia de productos mediante una estrategia sistemática e iterativa. Lo primero que se construye es un ejemplar que satisface los requisitos fijos de la familia. Después, se incorporan progresivamente capas de flexibilización que implementan los requisitos variables.

- Los requisitos fijos de una familia de productos suelen ser más estables que los requisitos variables. EDD separa la implementación de los requisitos fijos (el

ejemplar) de la implementación de los requisitos variables (los módulos que flexibilizan el ejemplar).

- La decisión de elaborar una familia a menudo se toma al detectar trabajo repetitivo en el desarrollo aislado de varios productos de un dominio o al identificar oportunidades de negocio en la ampliación de las prestaciones de un producto de éxito. EDD reconoce esta situación y trata de aprovecharla mediante la reutilización íntegra de un ejemplar.

Finalmente, en los capítulos 4, 5 y 6 se ha ilustrado la potencia de EDD y ETL con ejemplos de desarrollo de programas escritos en Java y C++; de procedimientos almacenados escritos en TRANSACT SQL; de juegos de prueba escritos en Java y Modula-2; y de documentación escrita en HTML y Javadoc.

7.2. Trabajo futuro

Pueden considerarse tres grandes líneas de trabajo futuro:

1. **La ampliación de EDD.** En la propuesta metodológica de esta tesis subyace una nueva organización del desarrollo de software basada en tres roles**:**

    - El "ingeniero del software clásico", que construye un ejemplar.

    - El "flexibilizador", que generaliza el ejemplar para satisfacer los requisitos de toda una familia de productos.

    - El "parametrizador", que obtiene productos específicos particularizando la flexibilización del ejemplar.

    Sería deseable que una futura versión de EDD profundizara en los detalles y la repercusión de este tipo de organización.

2. **La mejora tecnológica de ETL**. Para ello, podrían adoptarse las siguientes medidas:

    a. Desarrollo de un IDE que facilite la edición y depuración de generadores ETL.

    b. Implementación de ETL como técnica interna de un GPL. Aunque esto restrinja la capacidad de flexibilización de ETL al código de un GPL en particular, permitiría aprovechar las prestaciones del GPL para el control de errores (sistema de tipos…) en la escritura de los generadores.

    c. Definición de sustituciones más abstractas. Con el fin de incrementar el nivel de reutilización de los generadores, sería conveniente encontrar un mecanismo,

alternativo a las expresiones regulares, que permitiera expresar las sustituciones de una forma más abstracta e independiente del ejemplar.

d. Búsqueda de implementaciones alternativas para ETL. Por ejemplo, en la actualidad están apareciendo las primeras implementaciones de QVT; sería interesante examinar la capacidad de este estándar en lenguajes de transformación para flexibilizar ejemplares.

3. **La aplicación práctica de EDD y ETL al desarrollo de familias de productos en el ámbito industrial**. De esta aplicación, se derivarían resultados experimentales que facilitarían el perfeccionamiento y la divulgación de EDD y ETL. En este sentido, el Departamento de Ingeniería de Software y Sistemas Informáticos de la UNED está tramitando convenios con empresas privadas y organismos públicos para la aplicación de EDD y ETIL en sus factorías de software.

Bibliografía

[ACHL02] Asencio, A.; Cardman, S.; Harris, D.; Laderman, E. *Relating expectations to automatically recovered design patterns*. Ninth Working Conference on Reverse Engineering. 29 Oct.-1 Nov. 2002 Page(s):87 – 96.

[AEHH96] Andries, M.; Engels, G; Habel, A.; Hoffmann, B.; Kreowski, H.; Kuske, S.; Kuske, D.; Plump, D.; Schürr, A.; Taentzer, G. *Graph Transformation for Specification and Programming*. Technical Report 7/96, Universität Bremen, 1996, http://citeseer.nj.nec.com/article/andries96graph.html

[AISJ+77] Alexander, C.; Ishikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I.; Angel, S. *A Pattern Language*. Oxford University Press, Nueva York, 1977.

[AJPG06] The AspectJ Team. *The AspectJ Programming Guide*. 2006. http://www.eclipse.org/aspectj/doc/released/progguide/index.html

[AK02] Akehurst, D.; Kent, S. *A relational approach to defining transformations in a metamodel*. Fifth International Conference on the Unified Modeling Language – The Language and its Applications, vol. 2460 of LNCS, pp. 243–258. Springer-Verlag, Dresden, Germany, 2002.

[Ale03] Alexander, R. *The real costs of aspect-oriented programming*. IEEE Software, Volume 20, Issue 6, Nov.-Dec. 2003 Page(s):92 – 93.

[Alex01] Alexandrescu, A. *Modern C++ Design. Generic Programming and Design Patterns Applied*. Addison-Wesley 2001.

[AM06] *AndroMDA*. http://www.andromda.org/

[Amb00] Ambler, S. *Mapping objects to relational databases*. developerWorks (IBM's resource for developers). 2000. http://www-128.ibm.com/developerworks/library/ws-mapping-to-rdb/

[AS06] Interactive Objects. *ArcStyler*. http://www.interactive-objects.com/

[ASU90] Aho, A. V.; Sethi, R.; Ullman, J. D. Compiladores, Principios, técnicas y herramientas. Addison-Wesley Iberoamericana, 1990.

[AU06] *ArgoUML*. http://argouml.tigris.org/

[AV98] Appleby, D.; Vandekopple, J. J. *Lenguajes de programación. Paradigma y práctica*. McGraw-Hill, 1998.

[BA99] Boehm, B.; Abts, C. *COTS integration: plug and pray?*. Computer. Volume 32, Issue 1, Jan. 1999. Page(s):135-138.

[Bat04] Batory, D. *Feature-oriented programming and the AHEAD tool suite*. 26th International Conference on Software Engineering (ICSE 2004). 23-28 May 2004 Page(s):702 - 703.

[BB02] Barbeau, M.; Bordeleau, F. *A protocol stack development tool using generative programming*. ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6–8, 2002, LNCS 2487, Springer-Verlag (2002), pp. 93–109

[BCRW98] Batory, B.; Chen, G.; Robertson, E.; Wang, T. *Design Wizards and Visual Programming Environments for Generators*. Fifth International Conference on Software Reuse (Victoria, Canada, June 1998). IEEE Computer Society Press, pp. 255-267.

[BDJR03] Bézivin, J. Dupé, G.; Jouault, F.; Rougui, J. E. *First experiments with the ATL model transformation language: Transforming XSLT into XQuery*. In the online proceedings of the OOPSLA'03 Workshop on Generative Techniques in the Context of the MDA, http://www.softmetaware.com/oopsla2003/mda-workshop.html

[Bec02] Beck, K. *Una explicación de la Programación Extrema*. Addison Wesley, 2002.

[Bec99] Beck, K. *Embracing change with extreme programming*. Computer. Volume 32, Issue 10, Oct. 1999. Page(s):70 - 77

[Bem68] Bemer, R. W. *The economics of program production*. International Federation for Information Processing Congress (IFIP) 1968, Volume 2: Edinburgh, UK, Page(s):1626-1627

[Ber03] Bergsten, H. *JavaServer Pages*, 3rd Edition. O'Reilly, 2003.

[BF03] Balanyi, Z.; Ferenc, R. *Mining design patterns from C++ source code.* International Conference on Software Maintenance (ICSM 2003). 22-26 Sept. 2003 Page(s):305 – 314.

[BFP94] Bellinzona, R.; Fugini, M.G.; Pernici, B. *Reusing specifications in OO Applications.* IEEE Software, Volume 12, Issue 2, March 1995 Page(s):65 – 75.

[BG93] Biffl, S.; Grechenig, T. *Degrees of consciousness for reuse of software in practice: Maintainability, balance, standardization.* Seventeenth Annual Internacional Computer Software and Applications Conference, COMPSAC 93. 1-5 Nov. 1993. Page(s):107 – 114.

[BJY01] Bieman, J.M.; Jain, D.; Yang, H.J. *OO design patterns, design structure, and program changes: an industrial case study.* IEEE International Conference on Software Maintenance, 7-9 Nov. 2001 Page(s):580 – 589.

[BLM92] Brown, D.; Levine, J.; Mason, T. *lex & yac*c. O'Reilly Media, Inc.; 2nd edition, 1992.

[BM03] Braun, P.; Marschall, F. *The Bi-directional Object-Oriented Transformation Language.* Technical Report, Technische Universität München, TUM-I0307, May 2003.

[BMRSS96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-Oriented Software Architecture. A System of Patterns.* John Wiley & Sons, 1996.

[BO92] Batory, D. O'Malley, S. *The Design and Implementation of Hierarchical Software Systems with Reusable Componentes.* ACM Transactions on Software Engineering and Methodology, vol. 1, no. 4, October 1992, pp. 355-398.

[Boe88] Boehm, B. W. *A Spiral Model of Software Development and Enhancement.* IEEE Computer, Mayo 1988.

[Bra04] Bracha, G. *Generics in the Java Programming Language.* July 5, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

[Bre05] Breu, S. *Extending Dynamic Aspect Mining with Static Information.* Fifth IEEE

International Workshop on Source Code Analysis and Manipulation. 30-01 Sept. 2005 Page(s):57 – 65.

[BRJ99] Booch, G.; Rumbaugh, J.; Jacobson, I. *El Lenguaje Unificado de Modelado.* Addison-Wesley, 1999.

[BSR04] Batory, D.; Sarvela, J.N.; Rauschmayer, A. *Scaling step-wise refinement.* IEEE Transactions on Software Engineering, Volume 30, Issue 6, June 2004 Page(s):355 – 371.

[BSWM+03] Bieman, J.M.; Straw, G.; Wang, H.; Munger, P.W.; Alexander, R.T. *Design patterns and change proneness: an examination of five evolving systems.* Ninth International Software Metrics Symposium. 3-5 Sept. 2003. Page(s):40 – 49.

[CA06] Codagen. *Codagen Architect.* http://www.codagen.com/products/architect/default.htm

[CBUE02] Czarnecki, K.; Bednasch, T.; Unger, P.; Eisenecker, U.W. *Generative programming for embedded software: An industrial experience report.* ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6–8, 2002, LNCS 2487, Springer-Verlag (2002), pp. 156–172

[CD94] Cook, S.; Daniels, J. *Designing Object Systems: Object-Oriented Modeling with Syntropy.* Prentice Hall, 1994.

[CDI03] CBOP, DSTC, and IBM. *MOF Query/Views/Transformations, Revised Submission.* OMG Document: ad/03-08-03

[CE00] Czarnecki, K.; Eisenecker, U. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[CH03] Czarnecki, K.; Helsen, S. *Classification of Model Transformation Approaches.* 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, Anaheim, October 2003.

[Che98] Chechik, M. *SCR3: towards usability of formal methods.* Proceedings of CASCON'98, December 1998, pp. 177-191.

[Chi03] Chiang, C. *Towards software reuse using parameterized formal specifications.* IEEE

International Conference on Information Reuse and Integration, IRI 2003. Page(s):519 – 526

[CHU04] Czarnecki, K.; Helsen, S.; Eisenecker, U. *Staged Configuration Using Feature Models*. Software Product Lines Conference (SPLC). Boston, MA, USA, August 30-September 2, 2004, pp. 266-283.

[CHW98] Coplien, J.; Hoffman, D.; Weiss, D. *Commonality and Variability in Software Engineering*. IEEE Software, pp. 37-45, Dec. 1998.

[CLDG+05] Costagliola, G.; De Lucia, A.; Deufemia, V.; Gravino, C.; Risi, M. *Design Pattern Recovery by Visual Language Parsing*. Ninth European Conference on Software Maintenance and Reengineering (CSMR 2005). 21-23 March 2005 Page(s):102 – 111.

[Cle01] Cleaveland, J. C. *Program Generators with XML and Java*. Prentice Hall, 2001.

[CN99] Clements, P. *A Framework for Software Product Lines Practice*, Version 2.0. Report from the Product Line System Program, Software Engineering Institute, Pittsburg, PA, July 1999, www.sei.cmu.edu/plp

[Cop91] Coplien, J. O. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley Professional (August 30, 1991).

[Cop99] Coplien J. O. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[Cpp06] CppUnit. http://cppunit.sourceforge.net

[CR00] Cybulski, J. L; Reed, K. *Requirements Classification and Reuse: Crossing Domain Boundaries*. Sixth International Conference on Software Reuse. Vienna, Austria, June 27-29, 2000.

[CS95] Coplien, J. O.; Schmidt, D. C. *Pattern Languages of Program Design*. Addison-Wesley Professional, 1995.

[Cus89] Cusumano, M.A. *The software factory: a historical interpretation*. IEEE Software, Volume 6, Issue 2, Mar 1989 Page(s):23 – 30

[DE06] Software Engineering Institute. *Domain Engineering*. http://www.sei.cmu.edu/domain-engineering/domain_engineering.html

[Deu89] Deutsch, L. P. *Design reuse and frameworks in the Smalltalk-80 system*. In Ted J. Biggerstaff and Alan J. Perlis, editors, Software Reusability, Volume II: Applications and Experience, pages 57–71. Addison-Wesley, Reading, MA, 1989.

[DHO01] Demuth, B.; Hussmann, H.; Obermaier, S. *Experiments with XMI based transformations of software models*. In Workshop on Transformations in UML. 2001.

[Dij76] Dijkstra, E. W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[Dil94] Diller, A. *Z: An Introduction to Formal Methods*. John Wiley & Sons; 2 edition (June 16, 1994)

[DS05] Dallal, J. A.; Sorenson, P. *Reusing class-based test cases for testing object-oriented framework interface classes*. Journal of Software Maintenance and Evolution: Research and Practice. Volume 17, Issue 3, Date: May/June 2005, Pages: 169-196.

[DuB06] DuBois, P. Using the Ruby DBI Module. Document revision: 1.03. Last update: 2006-11-28. http://www.kitebird.com/articles/ruby-dbi.html

[DuC01] DuCharme, B. *XSLT Quickly*. Manning, 2001.

[Eck00] Eckel, B. *Thinking in C++, 2nd ed. Volume 1*. Prentice Hall, 2000.

[Eck03] Eckel, B. *Thinking in Patterns*. Revision 0.9, 5-20-2003. Capítulo titulado *Pattern refactoring*. http://www.mindview.net.

[Edw01] Edwards, S. H. *A framework for practical, automated black-box testing of component-based software*. Software Testing, Verification and Reliability. Volume 11, Issue 2, Date: June 2001, Pages: 97-111.

[EMF06] *Eclipse Modeling Framework*. http://www.eclipse.org/emf/

[ERB06] *ERB*. http://raa.ruby-lang.org/project/erb/

[EU06] Omondo. *EclipseUML* http://www.eclipseuml.com/

[FBBO+99] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. *Refactoring:*

Improving the Design of Existing Code. Addison-Wesley, 1999.

[FBFL05] Ferenc, R.; Beszedes, A.; Fulop, L.; Lele, J. *Design Pattern Mining Enhanced by Machine Learning*. 21st IEEE International Conference on Software Maintenance (ICSM'05). Page(s):295 – 304.

[FGDS06] France, R.B.; Ghosh, S.; Dinh-Trong, T.; Solberg, A. *Model-Driven Development Using UML 2.0: Promises and Pitfalls*. Computer. Volume 39, Issue 2, Feb. 2006 Page(s):59 – 66.

[FKGS04] France, R. B.; Kim, D.-K.; Ghosh, S.; Song, E. *A UML-based pattern specification technique*. IEEE Transactions on Software Engineering, Volume 30, Issue 3, March 2004. Page(s):193 – 206.

[Fow02] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[Fow03] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Third Edition. Addison Wesley, 2003.

[Fow05] Fowler, M. *Language Workbenches: The Killer-App for Domain Specific Languages?*. http://www.martinfowler.com/articles/languageWorkbench.html

[Fri02] Friedl, J. E. F. *Mastering Regular Expressions*. Second edition. O' Reilly, 2002.

[Gam91] Gamma, E. *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools*. Tesis Doctoral, Universidad de Zurich, Institut für Informatik, 1991.

[GFA98] Griss, M.; Favaro, J.; d' Alessandro, M. *Integrating feature modeling with the RSEB*. Fifth International Conference on Software Reuse (ICSR), IEEE Computer Society Press (1998), pp. 76–85.

[GHJV94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GI93] Girardi, M. R.; Ibrahim, B. *A software reuse system based on natural language specifications*. 5th Int. Conf. on Computing and Information. Sudbury, Ontario, Canada, p. 507-511, 1993.

[Gla90] Glassner, A. S. *Graphics Gems*. Morgan Kaufmann; Reissue edition (June 1, 1990).

[GR04] Gray, J.; Roychoudhury. *A Technique for Constructing Aspect Weavers Using a Program Transformation System*. International Conference on Aspect-Oriented Software Development (AOSD). Lancaster, UK, March 22-27, 2004, pp. 36-45.

[Gra04] Gray, J. et al. *Model-Driven Program Transformation of a Large Avionics Framework*. Generative Programming and Component Engineering (GPCE) 2004, LNCS 3286, pp. 361-378, 2004.

[GS04] Greenfield, J.; Short, K. *Software Factories: Assembling Applications with Patterns, Models, Framworks, and Tools*. Wiley, 2004.

[Har02] Harold, E. R. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*. Addison-Wesley, 2002.

[Hay04] Haywood, D. *MDA: Nice Idea, Shame About the…* The ServerSide.Com. http://www.theserverside.com/articles/article.tss?l=MDA_Haywood

[HC05] Huang, C.; Chen, W. Y. *A Semi-Automatic Generator for Unit Testing Code Files Based on JUnit*. IEEE International Conference on Systems, Man and Cybernetics. Volume 1, 10-12 Oct. 2005 Page(s):140 - 145.

[HEAC05] Heradio Gil, R; Estívariz López, J. F.; Abad Cardiel, I; Cerrada Somolinos, J. A. *Traducción de especificaciones a código ejecutable mediante transformadores de ejemplares*. V Jornadas sobre Programación y Lenguajes (PROLE'05), Granada, 14 a 16 de Septiembre de 2005. Páginas 185-191.

[Her03] Herrington, J. *Code Generation in Action*. Manning, 2003.

[HFR99] Harrison, N.; Foote, B.; Rohnert, H. *Pattern Languages of Program Design 4*. Addison Wesley Publishing Company, 1999.

[HML03] Heuzeroth, D.; Mandel, S.; Lowe, W. *Generating design pattern detectors from pattern specifications*. 18th IEEE International Conference on Automated Software Engineering. 6-10 Oct. 2003. Page(s):245 – 248.

[HUn06] http://httpunit.sourceforge.net

[IBM06] *IBM Patterns for e-Business web site.* http://www-128.ibm.com/developerworks/patterns/library/index.html

[IEEE1471] Institute of Electrical and Electronics Engineers, Inc. *IEEE 1471-2000.* http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html

[IS06] Intentional Software. http://intentsoft.com/

[Jam06] *The Jamda Project.* http://jamda.sourceforge.net/

[Jav06] Sun Microsystems. *Javadoc.* http://java.sun.com/j2se/javadoc/

[Jet06] JetBrains. *Meta Programming System.* http://www.jetbrains.com/mps/

[Jos04] *Jostraca 0.4.0.* http://www.jostraca.org/

[JS00] Jarzabek, S.; Seviora, R. *Engineering components for ease of customisation and evolution.* IEE Proceedings- Software. Volume 147, Issue 6, Dec. 2000, pp. 237 – 248.

[JUn06] JUnit. http://www.junit.org

[Jus96] Justo, J. L. B. *A repository to support requirement specifications reuse.* Information Systems Conference of New Zealand. 30-31 Oct. 1996 Page(s):53 – 62.

[KCHN+90] Kang, K.; Cohen, S.; Hess, J.; Nowak, W.; Peterson, S. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)

[KHHK+01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G. *An overview of AspectJ.* European Conference on Object Oriented Programming, 2001.

[Kim02] Kim H. *AspectC#: an AOSD implementation for C#.* M. Sc. Thesis, Comp. Sci., Trinity College, Dublin, Noviembre 2002.

[KK99] Klein, M; Kazman, R. *Attribute-Based Architectural Styles.* Technical Report CMU/SEI-99-TR-022. http://www.sei.cmu.edu/publications/documents/99.reports/99tr022/

99tr022abstract.html

[KMHC05] Kim, S. D.; Min, H. G.; Her, J.S.; Chang, S.H. *DREAM: a practical product line engineering using model driven architecture*. Third International Conference on Information Technology and Applications, (ICITA 2005). Volume 1, 4-7 July 2005. Page(s):70 – 75.

[Kru00] Kruchten, P. *The Rational Unified Process: An Introduc*tion, 2nd Edition. Addison Wesley, 2000.

[KRW02] Kirk, D.; Roper, M.; Wood, M. *Defining the problems of framework reuse*. 26th Annual International Computer Software and Applications Conference (COMPSAC 2002). 26-29 Aug. 2002. Page(s):623 – 626.

[KSBM99] Kwon, O. C.; Shin, G. S.; Boldyreff, C.; Munro, M. *Maintenance with reuse: an integrated approach based on software configuration management*. Sixth Asia Pacific Software Engineering Conference (APSEC '99). 7-10 Dec. 1999. Page(s):507 – 515.

[KSRP99] Keller, R.; Schauer R.; Robitaille, S.; Page, P. *Pattern-Based Reverse-Engineering for Design Components*. Twenty-First International Conference on Software Engineering, pages 226-235, Los Angeles, CA, May 1999.

[KWB03] Kleppe, A.; Warmer, J.; Bast, W. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[Lad03a] Laddad, R. *AspectJ in action*. Manning, 2003.

[Lad03b] Laddad, R. *Aspect-oriented programming will improve quality*. IEEE Software, Volume 20, Issue 6, Nov.-Dec. 2003 Page(s):90 – 91.

[Lar02] Larman, C. *Applying UML and Patterns—An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Second Edition. Prentice Hall, 2002.

[LG84] Lanergan, L. G.; Grasso, C. A. *Software Engineering with reusable designs and code*. IEEE Trans. Software Engineering, vol 10, no. 5, Sept. 1984. Page(s): 498-501.

[LGL02] Laguna, M. A.; García, F. J.; López, O. *Reutilización de Requisitos de Usuario: el Modelo Mecano*. Revista Colombiana de Computación, ISSN

1657 - 2831, Vol. 3:2, Diciembre 2002.

[LKL02] Lee, K.; Kang, K.C.; Lee, J. *Concepts and guidelines of feature modeling for product line software engineering*. Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), Austin, USA, Apr.15–19, 2002. LNCS 2319, Springer-Verlag (2002), pp. 62–77

[LL97] Lee, N. Y.; Litecky, C.R. *An empirical study of software reuse with special attention to Ada*. IEEE Transactions on Software Engineering, Volume 23, Issue 9, Sept. 1997. Page(s):537 – 549.

[LMV97] Lam, W.; McDermid, T.A.; Vickers, A.J. *Ten steps towards systematic requirements reuse*. Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on 6-10 Jan. 1997. Page(s):6 – 15.

[Lon98] Lonngren, D.D. *Reducing the cost of test through reuse*. AUTOTESTCON '98. IEEE Systems Readiness Technology Conference. 24-27 Aug. 1998 Page(s):48 – 53.

[Lou04] Louden, K. C. *Lenguajes de programación. Principios y práctica*. Thomson, 2004.

[MA02] Muthig, D.; Atkinson, C. *Model-Driven Product Line Architectures*. LNCS 2379, Proceedings of the 2nd Software Product Line Conference, 2002.

[Mat99] Mattsson, M. *Effort Distribution in a Six Year Industrial Application Framework Project*. IEEE Int'l Conf. Software Maintenance (ICSM '99), pp. 326-333, 1999.

[MB01] McNatt, W. B.; Bieman, J. M. *Coupling of design patterns: common practices and their benefits*. 25th Annual International Computer Software and Applications Conference (COMPSAC 2001). 8-12 Oct. 2001. Page(s):574 – 579.

[MB02] Mellor, S. J.; Balcer, M. J. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[MB97] Mattsson, M.; Bosch, J. *Framework composition: problems, causes and solutions*. Technology of Object-Oriented Languages and Systems, 1997. TOOLS 23. 28 July-1 Aug. 1997. Page(s):203 – 214.

[McI68] McIlroy, M. D. *Mass produced software components*. Proc. NATO Software Engineering Conference. Garmisch, Germany (1968), Page(s) 138-155.

[MCL04] Mak, J. K. H.; Choy, C. S. T.; Lun, D. P. K. *Precise modeling of design patterns in UML*. International Conference on Software Engineering (ICSE 2004). 26th 23-28 May 2004. Page(s):252 – 261.

[McN03] McNeile, A. *MDA: The Vision With the Hole?*. Metamaxim. http://www.metamaxim.com/download/documents/MDAv1.pdf

[MDA03] The Object Management Group. *MDA Guide Version 1.0.1*. 12th June 2003. http://www.omg.org/mda/

[Men01] Menard, J. *NQXML*. http://nqxml.sourceforge.net/

[Mey00] Meyer, B. *Object-Oriented Software Construction*. 2nd Edition. Prentice Hall, 2000.

[MF06] *ModFact*. http://forge.objectweb.org/projects/modfact/

[Mic97] Michael, C.C. *Reusing testing of reusable software components*. Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS '97). 'Are We Making Progress Towards Computer Assurance?'. 16-19 June 1997 Page(s):97 – 104.

[Mil02] Milicev, D. *Automatic model transformations using extended UML object diagramsin modeling environments*. IEEE Transactions on Software Engineering, vol. 28(4):pp. 413–431, 2002.

[Mil56] Miller, G. A. *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*. The Psychological Review, 1956, vol. 63, pp. 81-97

[Mil95] Mili, H.; Mili, F.; Mili, A. *Reusing software: issues and research directions*. IEEE Transactions on Software Engineering, Volume 21, Issue 6, June 1995. Page(s):528 – 562.

[Min06] Lego. *Mindstorms*. http://mindstorms.lego.com

[MKKW99] Mannion, M.; Keepence, B.; Kaindl, H.; Wheadon, J. *Reusing single system requirements from application family requirements*. International Conference on

Software Engineering. 16-22 May 1999 Page(s):453 – 462.

[MMWO94] von Mayrhauser, A.; Mraz, R.; Walls, J.; Ocken, P. *Domain based testing: increasing test case reuse*. IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. 10-12 Oct. 1994 Page(s):484 – 491.

[MN96] Moser, S.; Nierstrasz, O. *The Effect of Object-Oriented Frameworks on Developer Productivity*. Computer, pp. 45-51, Sept. 1996.

[MOF06] The Object Management Group. *Meta Object Facility Core Specification version 2.0*. OMG document formal/2006-01-01

[MRB97] Martin, R. C.; Riehle, D.; Buschmann, F. *Pattern Languages of Program Design 3*. Addison-Wesley Professional, 1997.

[MRS02] Morisio, M.; Romano, D.; Stamelos, I. *Quality, productivity, and learning in framework-based development: an exploratory case study*. IEEE Transactions on Software Engineering, Volume 28, Issue 9, Sept. 2002. Page(s):876 – 888.

[MSPB+00] Morisio, M.; Seaman, C.; Parra, A.; Basili, V.; Kraft, S.; Condon, S. *Investigating and improving a COTS-based software development*. 22nd International Conference on Software Engineering (ICSE), June 4-11, 2000. Limerick, Ireland. Pages: 32-41.

[MSSA+02] MacDonald, S.; Szafron, D.; Schaeffer, J.; Anvik, J.; Bromling, S.; Tan, K. *Generative design patterns*. 17th IEEE International Conference on Automated Software Engineering (ASE 2002). 23-27 Sept. 2002. Page(s):23 – 34.

[MSU98] Masuda, G.; Sakamoto, N.; Ushijima, K. *Applying design patterns to decision tree learning system*. Proc. ACM SICSOFT Int. Symp. Foundations of Sofware Engineering, pages 111-120, 1998.

[MSUW04] Mellor, S. J.; Scott, K.; Uhl, A.; Weise, D. *MDA Distilled. Principles of Model-Driven Architecture*. Addison-Wesley, 2004.

[MVN06] Manolescu, D; Voelter, M; Noble, J. *Pattern Languages of Program Design 5*. Addison-Wesley Professional, 2006.

[Mye89] Myers, W. *Allow Plenty of Time for Large-Scale Software*. IEEE Software, Volume 6, Issue 4, July 1989. Pages: 92-99.

[Nas06] Nasato, Y. *XMLParser*. http://www.yoshidam.net/Ruby.html#xmlparser

[NBMR06] *NetBeans Metadata Repository*. http://mdr.netbeans.org/

[Nei80] Neighbors, J. M. *Software Construction Using Components*. Tech. Report 160. Department of Information and Computer Sciences, University of California. Irvine, CA. 1980

[NUn06] NUnit. http://www.nunit.org

[ODM06] Software Engineering Institute. *Organization Domain Modeling*. http://www.sei.cmu.edu/str/descriptions/odm_body.html

[OJ06] Compuware. *OptimalJ*. http://www.compuware.com/products/optimalj/default.htm

[OT01] Ossher, H.; Tarr, P. *Multi-dimensional separation of concerns and the hyperspace approach*. Proc. of the Symposium on Software Architecture and Component Technology: The State of the Art in Software Development. Kluwer.

[Par72] Parnas, D. L. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, ACM Press, NewYork, NY, 1972, 15(12), pp. 1053-1058.

[Par76] Parnas, D. *On the Design and Development of Program Families*. IEEE Transactions on Software Engineering, March 1976.

[PBG01] Peltier, M.; Bézivina, J.; Guillaume, G. *MTRANS: A general framework, based on XSLT, for model transformations*. In J. Whittle et al. (eds.), Workshop on Transformations in UML, pp. 93–97. 2001.

[Pel02] Peltier, M.L. *MTrans, a DSL for model transformation*. Sixth International Enterprise Distributed Object Computing Conference (EDOC '02). 17-20 Sept. 2002. Page(s):190 – 199.

[PG02] Pinzger, M.; Gall, H. *Pattern-supported architecture recovery*. 10th International

Workshop on Program Comprehension. June 2002 Page(s):53 - 61

[PGram06] NorKen Technologies. *ProGrammar. Parser Development Toolkit.* http://www.programmar.com/

[PK98] Prechelt, L.; Kramer, C. *Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns.* Journal of Universal Computer Science 1998, pp. 866-882.gea

[Pod06] *POD, the Plain Old Documentation format.* http://perldoc.perl.org/perlpod.html

[Pos06] Gentleware. *Poseidon.* http://gentleware.com/index.php

[PUTB+01] Prechelt, L.; Unger, B.; Tichy, W.F.; Brossler, P.; Votta, L.G. *A controlled experiment in maintenance: comparing design patterns to simpler solutions.* IEEE Transactions on Software Engineering, Volume 27, Issue 12, Dec. 2001 Page(s):1134 – 1144.

[Pyt07] *The Python Programming Language.* http://www.python.org/

[PZ98] Pratt, T. W.; Zelkowitz, M. V. L*enguajes de programación. Diseño e implementación.* Prentice Hall, 1998.

[QVT05] The Object Management Group. *MOF QVT Final Adopted Specification.* OMG ptc/05-11-01

[Racc05] Racc. http://i.loveruby.net/en/projects/racc/

[Rdo05] *RDOC - Ruby Documentation System.* http://www.ruby-doc.org/stdlib/libdoc/rdoc/rdoc/index.html

[Ris00] Rising, L. *The Pattern Almanac 2000.* Addison Wesley Publishing Company, 2000.

[RJ00] Rising, L.; Janoff, N.S. *The Scrum software development process for small teams.* IEEE Software. Volume 17, Issue 4, July-Aug. 2000. Page(s):26 – 32.

[Rockit01] Rockit. http://rockit.sourceforge.net/

[RR06] IBM. *Rational Rose.* http://www-306.ibm.com/software/rational/

[Rub07] *Ruby Home Page*. http://www.ruby-lang.org/en/

[RUn06] RubyUnit. http://homepage1.nifty.com/markey/ruby/rubyunit/index_e.html

[Rus06] Russell, S. REXML. http://www.germane-software.com/software/rexml/

[RW04] Robinson, W.N.; Woo, H.G. *Finding reusable UML sequence diagrams automatically*. IEEE Software. Volume 21, Issue 5, Sep-Oct 2004. Page(s): 60-67

[Sch01] Schwanke, R.W. *Layers, decisions, patterns, styles, and architectures*. Working IEEE/IFIP Conference on Software Architecture. 28-31 Aug. 2001 Page(s):137 – 147.

[Sel05] Selby, R.W. *Enabling reuse-based software development of large-scale system*s. IEEE Transactions on Software Engineering, Volume 31, Issue 6, June 2005. Page(s):495 – 510.

[SF04] Shao, J.; Far, B. H. *Development of an Intelligent System for Architecture Design and Analysis*. IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2004), pp. 539-542, May 2004.

[SF06] Microsoft. *Software Factories*. http://msdn.microsoft.com/vstudio/teamsystem/workshop/sf/default.aspx

[SG96] Shaw, M; Garlan, D. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Sil03] Silva, A.R. *The XIS approach and principles*. 29th Euromicro Conference. 1-6 Sept. 2003. Page(s):33 – 40.

[Sim99] Simonyi, C. *The Future is Intentional*. Computer. Volume 32, Issue 5, May 1999 Page(s):56 – 57.

[SLB00] Shull, F.; Lanubile, F.; Basili, V.R. *Investigating Reading Techniques for Object-Oriented Framework Learning*. IEEE Trans. Software Eng., vol. 26, no. 11, Nov. 2000.

[SLU05] Spinczyk, O.; Lohmann, D.; Urban, M. *AspectC++: an AOP Extension for*

C++. Software Developer's Journal, pages 68-76, 05/2005.

[SMB96] Shull, F.; Melo, W. L.; Basili, V. R. *An Inductive Method for Discovering Desing Patterns from Object-Oriented Software Systems*. Technical report, 1996.

[Som05] Sommerville, I. *Ingeniería del Software*. Addison Wesley, 2005; 7ª edición.

[Spi06] Spinczyk, O. *AC++ Compiler Manual*. Version 1.1, March 15, 2006. http://www.aspectc.org

[SQVT07] *SmartQVT*. http://smartqvt.elibel.tm.fr

[Sri99] Srinivasan, S. *Design patterns in object-oriented frameworks*. Computer. Volume 32, Issue 2, Feb. 1999. Page(s):24 – 32.

[SSRG+05] Solberg, A.; Simmonds, D.; Reddy, R.; Ghosh, S.; France, R. *Using aspect oriented techniques to support separation of concerns in model driven development*. 29th Annual International Computer Software and Applications Conference (COMPSAC 2005). Volume 1, 26-28 July 2005 Page(s):121 - 126 Vol. 2.

[Szy02] Szyperski, C. *Component Software*. Addison-Wesley; 2nd edition. 2002.

[TArch06] *Borland Together Architect 2006*. http://www.borland.com/us/products/together

[TH01] Thomas, D.; Hunt, A. *Programming Ruby. The Pragmatic Programmers' Guide*. Addison Wesley, 2001.

[TL03] Taibi, T.; Ling, D. N. C. *Formal specification of design patterns: a comparison*. ACS/IEEE International Conference on Computer Systems and Applications. Book of Abstracts. 14-18 July 2003.

[TMQH+03] Trowbridge, D; Mancini, D.; Quick, D.; Hohpe, G.; Newkirk, J.; Lavigne, D. *Enterprise Solution Patterns Using Microsoft .NET*. Microsoft Corporation, June 2003. http://msdn.microsoft.com/practices/compcat/default.aspx?pull=/library/en-us/dnpatterns/html/esp.asp

[TPSG+02] Teboul, L.; Pawlak, R.; Seinturier, L.; Gressier-Soudan, E.; Becquet, E. *AspectTAZ : a new approach based on aspect oriented programming for object oriented industrial messaging services design*. 4th IEEE International Workshop

on Factory Communication Systems, 2002. Page(s):165 – 171

[Tra94] Tracz, W. *Software reuse myths revisite*d. 16th International Conference on Software Engineering (ICSE-16) 16-21 May 1994. Page(s):271 – 272.

[TRHM+04] Trowbridge, D.; Roxburgh, U.; Hohpe, G.; Manolescu, D.; Nadhan, E. G. *Integration Patterns*. Microsoft Corporation, June 2004. http://msdn.microsoft.com/practices/compcat/default.aspx?pull=/library/en-us/dnpag/html/intpatt.asp

[TSS04] Turki, S.; Soriano, T.; Sghaier, A. *An MDA application for a virtual reality environment*. IEEE International Conference on Industrial Technology (IEEE ICIT '04). Volume 2, 8-10 Dec. 2004. Page(s):807 – 812.

[TXL06] *The TXL Programming Language*. http://www.txl.ca/

[UML20] The Object Management Group. *Unified Modeling Language: Superstructure, Version 2.0*. OMG document formal/05-07-04, 2004.

[VCK96] Vlissides, J. M.; Coplien, J. O.; Kerth, N, L. *Pattern Languages of Program Design 2*. Addison-Wesley Professional, 1996.

[Vok04] Vokac, M. *Defect frequency and design patterns: an empirical study of industrial code*. IEEE Transactions on Software Engineering, Volume 30, Issue 12, Dec. 2004 Page(s):904 – 917.

[VSt05] Microsoft. *Visual Studio 2005*. http://msdn.microsoft.com/vstudio/

[VV00] Viega, J.; Vuas, J. *Can aspect-oriented programming lead to more reliable software?* IEEE Software, Volume 17, Issue 6, Nov.-Dec. 2000 Page(s):19 – 21.

[VVP02] Varro, D.; Varro, G.; Pataricza, A. *Designing the automatic transformation of visual languages*. Science of Computer Programming, vol. 44(2):pp. 205--227, 2002.

[WBM99] Walker, R.J.; Baniassad, E.L.A.; Murphy, G.C. *An initial assessment of aspect-oriented programming*. International Conference on Software Engineering. 16-22 May 1999 Page(s):120 – 130.

[Weg78] Wegner, P. *Research directions in software technology*. 3rd International Conference on Software Engineering, 1978.

[Wei00] Weissinger, K. *ASP in a Nutshell*, 2nd Edition. O'Reilly, 2000.

[Wir71] Wirth, N. *Program development by stepwise refinement*. Communications of the ACM. Volume 14, Issue 4 (April 1971). Pages: 221 - 227.

[Wir76] Wirth, N. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.

[WK03] Warmer, J.; Kleppe, A. *The Object Constraint Language: Getting Your Models Ready for MDA*. Second Edition. Addison Wesley, 2005.

[WL99] Weis, D. M.; Lai, C. T. R. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[WT05] Wang, W.; Tzerpos, V. *Design Pattern Detection in Eiffel Systems*. 12th Working Conference on Reverse Engineering, 07-11 Nov. 2005 Page(s):165 – 174.

[WZZ03] Wang, H.; Zhang, D.; Zhou, J. *MDA-based development of e-learning system*. 27th Annual International Computer Software and Applications Conference (COMPSAC 2003). 3-6 Nov. 2003. Page(s):684 – 689.

[Xac07] *Xactium*. http://www.xactium.com

[XMI05] The Object Management Group. *XML Metadata Interchange (XMI), v2.1*. OMG document formal/2005-09-01

[YA03] S. M. Yacoub, H. H. Ammar. *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley, 2003.

[YC79] Yourdon, E.; Constantine, L. L. Structured Design. *Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979.

[YLM04] Yu, Y.; Leite, J. C. S. P.; Mylopoulos, J. *From goals to aspects: discovering aspects from requirements goal models*. 12th IEEE International Requirements Engineering Conference, 2004. Page(s):38 – 47.

[YXA00a] Yacoub, S. M.; Xue, H.; Ammar, H. H. *Automating the development of pattern-oriented designs for application specific software systems*. 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology. 24-25 March 2000. Page(s):163 – 170.

[YXA00b] Yacoub, S. M.; Hengyi Xue; Ammar, H. H. *POD: a composition environment for pattern-oriented design*. 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34). 30 July-4 Aug. 2000. Page(s):263 – 272.

[Zac04] Zachman, J. *Enterprise Architecture Framework*. http://www.zifa.com/

[ZBJ04] Zhu, L.; Babar, M. A.; Jeffery, R. *Mining patterns to support software architecture evaluation*. Fourth Working IEEE/IFIP Conference on Software Architecture.
12-15 June 2004 Page(s):25 – 34.

[Zha05] Zhang, G. *Towards aspect-oriented class diagrams*. 12th Asia-Pacific Software Engineering Conference (APSEC '05).15-17 Dec. 2005 Page(s):6 pp.

[ZLB04] Zhang, Z.; Li, Q.; Ben, K. *A new method for design pattern mining*. International Conference on Machine Learning and Cybernetics. Volume 3, 26-29 Aug. 2004 Page(s):1755 – 1759.