UNIVERSIDADE
DE VIGO

Dpto. de Enxeñería Telemática
ETSE de Telecomunicación

TESIS DOCTORAL

SEPARACIÓN MULTIDIMENSIONAL DE ASPECTOS PARA LA ESPECIFICACIÓN Y REUTILIZACIÓN DE REQUISITOS EN LENGUAJE Z

Autor: **Ramón Ventura Roque Hernández**
Ingeniero en Sistemas Computacionales

Directoras: **Rebeca P. Díaz Redondo**
Doctor Ingeniero de Telecomunicación
Ana Fernández Vilas
Doctor Ingeniero de Telecomunicación

Junio, 2011

Agradecimientos

A Dios, por jamás olvidarse de mí y bendecirme en abundancia.

A mi madre, por estar siempre presente a mi lado, apoyándome.

A mis asesoras de tesis, Dra. Rebeca Díaz Redondo y Dra. Ana Fernández Vilas de la Universidad de Vigo por todo el apoyo que me han brindado. Sin sus valiosos consejos, comentarios, aportaciones y correcciones, no hubiera sido posible llevar a cabo este trabajo.

A todos mis maestros de la Universidad de Vigo, del Instituto Tecnológico de Nuevo Laredo y de todas las instituciones en donde he estudiado. Sus enseñanzas han sido invaluables para mí. Su ejemplo ha contribuido positivamente a mi vida profesional.

A todos mis amigos, y ex compañeros de estudio de todo el recorrido educativo de mi vida.

A todos mis amigos y compañeros del Doctorado en Educación del Instituto de Estudios Superiores José Martí, porque junto a ellos estoy recorriendo una nueva aventura pedagógica que ha enriquecido mi aprendizaje.

A mis compañeros de trabajo, directivos y amigos de la Facultad de Comercio, Administración y Ciencias Sociales de la Universidad Autónoma de Tamaulipas, por su confianza en mí.

A mis compañeros de trabajo, directivos y amigos del Instituto Tecnológico de Nuevo Laredo, por esa confianza, calidez y aceptación que me han brindado.

A todos mis alumnos, y ex alumnos, porque siempre me motivan a superarme día a día. Por ellos me he dado cuenta de lo importante que es aprender siempre y de la gran responsabilidad que tengo al ser su facilitador en las asignaturas; por ellos he encontrado fascinante el mundo de la docencia. Gracias por permitirme seguir aprendiendo a su lado.

A todos los que me han influenciado positivamente, a los que están en mi “aquí y ahora”, a los que ya se adelantaron en el camino y con su ausencia han fortalecido su presencia en mi vida. A todos aquellos que recuerdo con cariño por haber vivido conmigo anécdotas entrañables, gracias, muchas gracias.

Ramón
Junio, 2011.

Resumen

La separación de intereses (*Separation of Concerns*) ha sido abordada desde hace varias décadas como una manera de enfrentar la creciente complejidad en los sistemas de software. Un interés (*concern*) es una parte o unidad que es relevante para un concepto particular, objetivo o propósito relacionado con el software que se está desarrollando. En la Ingeniería de Requerimientos se ha reconocido la importancia de contar con una adecuada separación de intereses para lograr una buena gestión de las especificaciones y contar con características como buena trazabilidad, facilidad de mantenimiento y evolución en los elementos subsecuentes a lo largo de todas las actividades de desarrollo.

La separación de intereses en el paradigma orientado a objetos ha resultado útil para enfrentar la complejidad en la construcción de software. Las clases proporcionan grandes ventajas al permitir clasificar los intereses en jerarquías; sin embargo, es bien reconocido que este paradigma puede resultar inflexible pues impone una descomposición de clases que es adecuada para una sola separación de intereses en particular. Esto impide manejar efectivamente los intereses transversales (aquellos que impactan el software en diferentes lugares).

Por su parte, la orientación a aspectos (OA) es una forma de separar los intereses de un sistema de software que pretende superar las desventajas de la separación de intereses de la orientación a objetos. En la última década la OA ha sido bien recibida en la fase de programación de aplicaciones y ha ganado popularidad principalmente gracias a la existencia de un lenguaje orientado a aspectos estable y accesible como ASPECT-J basado en Java. ASPECT-J ha implementado un modelo de separación de intereses bidimensional, en donde la funcionalidad principal del software se aísla de los intereses transversales; estos últimos se encapsulan en aspectos y contienen información esencial para afectar a la funcionalidad base.

El modelo bidimensional ha sobrepasado la fase de programación para influenciar los diferentes enfoques de trabajo de las primeras etapas del ciclo de vida, en los cuales se ha acuñado el término de "aspectos tempranos" para referirse a los intereses transversales durante las etapas previas a la programación. A pesar de su popularidad, el modelo bidimensional tiene limitaciones reconocidas, debidas a que no supera totalmente las limitantes impuestas por la descomposición inicial dominante de intereses establecida por los paradigmas tradicionales como el orientado a objetos; por ejemplo, en este modelo los aspectos mantienen un fuerte acoplamiento con los intereses base, lo que dificulta su reutilización, reduce la facilidad de configuración y disminuye la versatilidad que el desarrollador puede lograr con el software.

Como una solución a estos problemas del modelo bidimensional, surgió el modelo abstracto multidimensional que puede ser adaptado a las diferentes etapas del ciclo de vida para trabajar desde diversas perspectivas, según sean las necesidades particulares del formalismo que lo adopte. El objetivo principal del modelo multidimensional es superar las limitaciones de adherirse a una descomposición inicial proporcionando, en consecuencia,

las ventajas de mejor modularización, reutilización y facilidad de configuración y mantenimiento.

En un espacio influenciado por el modelo multidimensional, en el presente trabajo convergen cuatro áreas de estudio de la Ingeniería del Software: separación de intereses, aspectos tempranos en la Ingeniería de Requerimientos, lenguajes formales y reutilización. En esta tesis se propone Z-SIIMD (Z con Separación de Intereses con Influencia Multidimensional), un enfoque de trabajo para la especificación de requerimientos formales en lenguaje Z tomando como base una separación multidimensional de intereses que permita contar con un escenario donde sea posible reutilizar los elementos de las especificaciones realizadas.

Con la influencia del modelo multidimensional, particularmente para la etapa de especificación de requerimientos, en esta tesis se ha optado por no exigir una organización inicial única de intereses dividida en funcionales y no funcionales o transversales, por el contrario, todos los intereses se pueden especificar de manera homogénea con sus estados, entradas y salidas. Posteriormente, mediante reglas separadas se establecen las relaciones entre los intereses con sus respectivos detalles de afectación.

Las reglas contienen las directrices que guían al proceso de composición, el cual se ocupa de construir un modelo integrado completo en base a los requerimientos especificados y a las reglas definidas en un proyecto. Las reglas permiten organizar mejor las interacciones entre los intereses y hacen que su análisis resulte más sencillo; también facilitan el mantenimiento general de las especificaciones al mismo tiempo que dejan sin cambio la estructura original de los intereses, aún cuando las relaciones entre ellos se modifiquen.

Se propone también el uso de un repositorio para almacenar requerimientos, intereses y reglas de composición con el objetivo de sustentar su reutilización. De esta manera, los elementos de un proyecto pueden ser agregados al repositorio, o de él pueden ser extraídos algunos elementos existentes para incorporarlos al diseño de cualquier proyecto en curso. Para facilitar su clasificación y búsqueda, se proporcionan palabras clave o definiciones en lenguaje natural para cada elemento en el proyecto.

Con el objetivo de validar la propuesta, se ha desarrollado un prototipo software denominado SPZS (Software Prototipo para Z-SIIMD), que implementa las principales características de esta tesis, proporcionando los medios para capturar requerimientos, intereses y reglas y obtener un modelo completo configurable después de haber ejecutado un algoritmo de composición. El prototipo realizado también incluye un mecanismo de reutilización que facilita el acceso a un repositorio de elementos que pueden ser incorporados en diferentes proyectos de especificación.

En conclusión, en esta tesis se propone una manera de especificar formalmente los requerimientos de software en lenguaje Z desde una perspectiva multidimensional que propicia la reutilización de los elementos involucrados en las especificaciones, y se acompaña de un prototipo de software que incorpora las principales ideas del trabajo realizado.

Abstract

Separation of concerns has been studied for many decades as a way to face the increasing complexity of software systems. A concern is defined as a relevant unit of a particular concept related to the software being developed. Requirements Engineering has identified the relevance of having an adequate separation of concerns in order to achieve in specifications desirable properties such as: good administration, trazability, maintainability and evolution of the later artifacts along the software life cycle activities.

Object Orientation (OO) proposes a particular way to separate concerns that has been useful to handle the complexity during software development. Classes offer great advantages when it comes to classify concerns in hierarchies; nevertheless, this paradigm falls short in handling crosscutting concerns, which are defined as those affecting the software in different places.

Aspect Orientation (AO), on the other hand, is another way of separating concerns in a software system that aims to overcome the disadvantages of OO. Over the last decade, AO has been used at the programming stage, and has become popular because of ASPECT-J, a very stable and easy-to-get Java-Based Aspect-Oriented Language. ASPECT-J implements separation of concerns from a bi-dimensional perspective, where software's main functionality is isolated from the crosscutting concerns. The crosscutting concerns encapsulate aspects and contain the information needed to affect the main functionality.

Bi-dimensional model has surpassed the programming stage, and has influenced different approaches in the early software life cycle activities, where the term "early aspects" has been coined. Early aspects refer to crosscutting concerns in any stage before programming. Despite its popularity, Bi-dimensional model has known issues because it requires an initial decomposition which dominates the rest of the development process. Aspects are coupled with the main concerns, and this result in difficult reuse, reduced configuration possibilities and limited developer's versatility.

As a solution to the problems imposed by the bi-dimensional model, the multidimensional abstract model was proposed. It can be adapted to different life cycle activities according to the particular needs of the methodology being used. The main objective of the multi-dimensional model is to offer a better separation of concerns unattached from an initial decomposition. As a consequence, better modularization, reuse, ease of configuration and maintenance can be achieved.

This thesis is influenced by the multi-dimensional model; it illustrates work on the following four areas: Separation of Concerns, Early Aspects in Engineering Requirements, Formal Languages, and Specification Reuse.

This thesis proposes Z-SIIMD (Z con Separación de Intereses con Influencia Multidimensional – Multi-Dimensional Influenced Separation of Concerns in Z), an

approach to specify formal requirements using Z. It is based on a Multi-Dimensional Separation of Concerns where specifications reuse can be achieved easily.

Z-SIIMD does not require initial requirements specifications to be categorized into "Functional" and "Non-Functional"; on the other hand, every requirement can be specified homogeneously containing inputs, outputs and states. After the requirements specification, isolated composition rules allow establishing relationships among requirements; these rules contain all the details to accomplish this task.

Composition rules guide a composition process that generates an integrated model based on the specified requirements and the rules contained on a project. Rules allow organizing requirements and maintaining specifications in an efficient way, while leaving the original structure of the requirements unchanged, even if the relationships among them change.

This thesis also proposes the use of a repository to store requirements, concerns and composition rules in order to support specification reuse. Elements in a project can be added to the repository or they can be retrieved from it to incorporate them in a different project. Classification and matching elements in the repository can be made using keywords or natural language definitions.

Although the release of a final software product was never intended, a prototype for a software tool was developed as a way to validate this work. It was named SPZS which stands for "Software Prototipo para Z-SIIMD" (Prototype Software for Z-SIIMD). SPZS is a particular implementation of the main ideas covered on this thesis. It offers a graphical interface to type requirements, concerns and rules; it also generates and displays a configurable integrated model based on the typed elements in the project. SPZS also offers a reuse repository that can be accessed from any project.

This thesis proposes an approach to specify formal software requirements using Z from a multi-dimensional perspective that supports specification reuse. A software prototype tool was also developed to implement the main topics covered on this work.

CONTENIDO

Capítulo 1. Introducción 1
1.1. Antecedentes 1
1.2. Motivación 3
1.3. Objetivos y contribuciones del presente trabajo 6
1.4. Organización 8
Parte I. Estado del arte 9
Capítulo 2. Métodos Formales 11
2.1. ¿Qué son los Métodos Formales? 11
2.2. Origen 11
2.3. Ventajas y desventajas 12
2.4. Taxonomía de las especificaciones formales 14
2.4.1. Especificación orientada a comportamientos 14
2.4.2. Especificación orientada a estados 15
2.4.3. Especificación orientada a transiciones 15
2.4.4. Especificación funcional 16
2.4.4.1. Especificación algebraica 16
2.4.4.2. Especificación con funciones de orden superior 16
2.4.5. Especificación operacional 16
2.5. Los métodos formales en las diferentes actividades del ciclo de vida 16
2.5.1. Los métodos formales en la especificación de requerimientos 18
2.6. Enfoques para la especificación formal 20
2.7. Descripción de algunos lenguajes formales 22
2.8. Conclusiones 29
Capítulo 3. Reutilización 31
3.1. ¿Qué es la reutilización de software? 31
3.1.1. Breve evolución histórica de la reutilización de software 31
3.1.2. Motivación para la reutilización 32
3.1.3. La paradoja de la administración de la reutilización de software 33
3.1.4. Tipos de reutilización 34
3.2. Reutilización por composición 35
3.3. Reutilización por generación 37
3.4. Tendencias en la reutilización 39
3.4.1. *Component Based Software Engineering* (CBSE) 39
3.4.1.1. Componentes 40
3.4.1.2. Modelos de componentes 41
3.4.1.3. Ciclo de vida CBSE 42
3.4.2. *Product Line Engineering* (PLE) 43
3.4.2.1. Ciclo de vida PLE 44
3.4.3. *Component Off the Shelf Based Development* (COTS) 47
3.4.3.1. Ciclo de vida COTS 47
3.5. Conclusiones 48
Capítulo 4. Separación de intereses y orientación a aspectos 51
4.1. Separación de intereses 51
4.1.1. Breve evolución histórica de la separación de intereses 53
4.2. Perspectivas para el estudio de la separación de intereses orientada a aspectos 54
4.2.1. Modelo de separación aspectual bidimensional de intereses 55
4.2.2. Modelo de separación aspectual multidimensional de intereses 57
4.3. Beneficios de la separación de intereses orientada a aspectos 61
4.4. Problemas de la separación de intereses orientada a aspectos 62
4.5. Conclusiones 63
Capítulo 5. Aspectos tempranos 65
5.1. Introducción 65

5.2. Aspectos en el análisis de dominio....66
5.3. Aspectos en la ingeniería de requerimientos....67
5.4. Aspectos en la arquitectura....72
5.5. Aspectos en el diseño....74
5.6. Conclusiones....77
Parte II. El trabajo de tesis....79
Capítulo 6. Presentación....81
6.1. Introducción....81
6.2. Requerimientos e intereses....82
6.3. Reglas de composición....83
6.4. El proceso de composición....84
6.5. El proceso de reutilización....84
6.6. El software SPZS....85
6.7. Organización....85
Capítulo 7. Elementos del modelo de trabajo....87
7.1. Introducción....87
7.2. Modelo de los elementos....88
7.3. Modelo para una regla de composición....90
7.3.1. Opciones de las reglas de composición....92
7.4. Sintaxis....99
7.4.1. Modelo base....99
7.4.2. Requerimiento....100
7.4.3. Interés....101
7.4.4. Regla de composición....102
7.4.5. Proyecto....103
7.4.6. Símbolos definidos en Z tradicional....104
7.5. Conclusiones....104
Capítulo 8. Modelo de composición....105
8.1. Introducción....105
8.2. Modelo de composición....105
8.3. Algoritmo de composición....108
8.3.1. Objetivo del algoritmo....108
8.3.2. Diagrama de alto nivel....109
8.3.3. Pseudocódigo....110
8.4. Conclusiones....117
Capítulo 9. Modelo de reutilización....119
9.1. Introducción....119
9.2. Modelo de reutilización....119
9.2.1. Clasificación....120
9.2.2. Instanciación....122
9.2.3. Reflexión sobre la clasificación y búsqueda de elementos....123
9.3. Algoritmo de reutilización....123
9.3.1. Objetivo del algoritmo....123
9.3.2. Diagramas de alto nivel....124
9.3.3. Pseudocódigo....126
9.4. Conclusiones....130
Capítulo 10. El prototipo de software....133
10.1. Introducción....133
10.2. Análisis y diseño del prototipo....133
10.3. Funcionalidad del prototipo....135
10.4. Alcance del prototipo....141
10.5. Conclusiones....143
Capítulo 11. Aplicación del proceso de composición de especificaciones. Caso de uso: agrupamiento de conexiones....145

11.1. Introducción 145
11.2. Modelado de agrupamiento de conexiones 146
11.3. Obteniendo conexiones del agrupamiento 147
11.4. Agregando conexiones al agrupamiento 148
11.5. Modelando separadamente una bitácora de actividades 149
11.6. Agregando el registro de bitácora a las operaciones del agrupamiento de conexiones 150
11.6.1. Registrando en la bitácora la obtención correcta de una conexión 150
11.6.2. Registrando en la bitácora el intento fallido de obtención de una conexión 151
11.6.3. Registrando en la bitácora la colocación exitosa de una conexión 152
11.6.4. Registrando en la bitácora el intento fallido de colocación de una conexión 152
11.7. Ejemplo de ejecución del algoritmo de composición 153
11.8. El modelo compuesto completo 156
11.9. Conclusiones 161
Capítulo 12. Aplicación de los procesos de clasificación e instanciación. Caso de uso: Retiro bancario 163
12.1. Introducción 163
12.2. Fase de clasificación 163
12.3. Fase de instanciación (en un proyecto diferente) 165
12.4. Agregando registro de bitácora a las operaciones de retiro 170
12.5. El modelo compuesto 172
12.6. Conclusiones 175
Parte III. Conclusiones 177
Capítulo 13. Conclusiones y líneas de trabajo futuro 179
13.1. Conclusiones 179
13.1.1. Especificación formal en lenguaje Z 179
13.1.2. Reutilización 180
13.1.3. Separación de intereses y aspectos tempranos 180
13.1.4. Implementación 181
13.2. Líneas de trabajo futuro 181
13.2.1. Continuación del trabajo 182
13.2.2. Líneas complementarias 183
Apéndice: Bases de Lenguaje Z 185
Capítulo 14. Bases del lenguaje Z 187
14.1. ¿Qué es el lenguaje Z? 187
14.2. Tipos 187
14.3. Conjuntos 188
14.4. Operadores de conjuntos 189
14.5. Declaraciones y variables 189
14.6. Variables globales y definiciones axiomáticas 190
14.7. Constantes y definiciones abreviadas 191
14.8. Tuplas y relaciones 192
14.9. Pares 193
14.10. Relaciones binarias 193
14.11. Operadores para las relaciones binarias 194
14.12. Funciones 195
14.13. Secuencias 197
14.14. Operadores de secuencias 198
14.15. Conectivas Lógicas 199
14.16. Cuantificadores 201
14.17. Esquemas 202
14.17.1. Cálculo de esquemas 206
14.17.2. Observaciones para la combinación de esquemas 209
14.17.3. Renombrando variables 210
14.17.4. Ocultando variables 211

14.17.5. Composición de esquemas....211
14.17.6. Piping....213
Bibliografía....215

Capítulo 1. Introducción

1.1. Antecedentes

La importancia de una buena separación de intereses se ha abordado desde hace varias décadas como una forma de enfrentar la complejidad (PARNAS, D. L., 1972). Sin embargo, con el advenimiento de la orientación a aspectos en la programación y su dispersión a las primeras actividades del desarrollo de software, la separación temprana de intereses ha adquirido mayor relevancia en el desarrollo de sistemas y se han destacado aún más las características de flexibilidad, facilidad de comprensión, configuración y reutilización que proporciona a los sistemas (COLYER, Adrian et al., 2004), (YU, Huiqun et al., 2005).

La orientación a aspectos se ha utilizado exitosamente en la etapa de implementación; sin embargo, es imprescindible incluirla también en las primeras actividades del ciclo de vida, por ejemplo en la especificación de requerimientos, desde donde la identificación de los aspectos tempranos se ha encontrado particularmente útil (NUSEIBEH, Bashar, 2004).

Al trabajar con aspectos de manera uniforme durante todo el proceso de desarrollo, se pretende manejar correctamente los requerimientos transversales desde un principio y evitar la incorporación de inconsistencias o relaciones forzadas al mezclar tareas orientadas a aspectos con otras que no lo son. Por ejemplo, los requerimientos funcionales transversales abordados desde una perspectiva puramente orientada a objetos, conllevan a decisiones de adecuación de arquitectura que afectan la calidad general del sistema (MOREIRA, Ana et al., 2005). Otro caso es en el subsecuente paso de un diseño tradicional a una implementación con aspectos, donde el programador debe adaptar el diseño realizado para incluir detalles aspectuales, terminando en tareas adicionales, diseños confusos y no concordantes con el código.

La separación temprana de intereses aspectuales se ha abordado principalmente desde dos perspectivas: bidimensional y multidimensional (apartado 4.2). La bidimensional consiste en realizar la especificación de los intereses base y, separadamente, la de los transversales (aspectuales) en función de los intereses base; finalmente mediante un mecanismo de combinación se unifican ambos tipos de intereses.

Por el contrario, desde una perspectiva multidimensional instanciando el modelo abstracto propuesto por (TARR, Peri and Sutton, Jr., Stanley M., 1999), es posible especificar los intereses en general de la manera más modular posible sin importar si son transversales o no. Posteriormente mediante un proceso de integración, se mezclan entre sí y dan lugar a nuevos módulos compuestos.

La mayoría de los enfoques de especificación de requerimientos aspectuales propuestos y utilizados hasta ahora se basan en la perspectiva bidimensional, la cual provoca que un solo criterio de descomposición dominante inicial guíe el proceso, dividiendo a los intereses en dos categorías: base y transversales. Sin embargo, algunas veces resulta prematuro pensar en una separación de esta naturaleza durante la etapa de especificación de requerimientos, pues no hay aún muchas directrices para categorizar adecuadamente los intereses sobre una descomposición dominante (DÍAZ PACE, J. Andrés and R. Campo, Marcelo, 2001). Además, el modelo bidimensional conlleva a una serie de problemas que involucran la dificultad de reutilización, la dependencia entre los intereses aspectuales y base, y la falta de flexibilidad al trabajar con ellos. Aunque se han realizado propuestas para eliminar algunos de esos inconvenientes (ROBERTS, Nathan V. et al., 2009), siempre se deben incorporar elementos adicionales al escenario de trabajo aspectual que podrían incrementar el grado de complejidad.

El modelo multidimensional ofrece una manera más aislada de separar los intereses al mismo tiempo que supera en gran medida los problemas de utilizar una sola descomposición dominante, propuesta por el modelo bidimensional. Si se agrega el uso de un lenguaje formal y características de reutilización al escenario de la especificación de requerimientos con influencia multidimensional para soportar la orientación a aspectos, se tiene un marco de trabajo viable para representar formalmente requerimientos, transversales o no, mantenerlos adecuadamente y aprovechar el esfuerzo invertido en las especificaciones realizadas.

Los lenguajes formales se han utilizado exitosamente para realizar especificaciones precisas de los sistemas. Su principal ventaja al incorporarlos al ciclo de vida es que permiten obtener un software más apegado a los requisitos planteados desde un principio. Su uso conjuntamente con estrategias de verificación y validación es esencial para lograr este resultado. Introducir formalidad en la ingeniería de requisitos ayuda a que las especificaciones sean más completas y robustas; es así como al tener requerimientos mejor capturados, la buena calidad del software es más fácil de lograr. El objetivo principal a final de cuentas es producir software demostrablemente correcto y libre de errores (DILLER, Antoni, 1999). Por otra parte, el uso de los métodos formales requiere de tiempo para aprenderlos y aplicarlos correctamente, de herramientas de software especializadas y de personal entrenado, además de que deben estar integrados en el ciclo de desarrollo del sistema. A pesar de su gran aplicabilidad, estas condiciones han propiciado su uso limitado en el desarrollo de software comercial cotidiano.

En la actualidad hay una marcada tendencia para utilizar los lenguajes formales existentes y adecuarlos para incorporar aspectos; con esto se conservan las características de los mejores lenguajes de especificación y se evitan agregaciones innecesarias de complejidad al estado del arte (NAFEES QAMAR, N. et al., 2006).

Un lenguaje formal existente que ha ganado bastante popularidad en los medios académicos, en los diferentes campos de aplicación en proyectos reales, y en la bibliografía en general es el lenguaje Z. En la práctica se han realizado especificaciones en lenguaje Z de sistemas médicos, nucleares, ferroviarios, y de comunicación, por mencionar algunos

ejemplos. En los textos clásicos de ingeniería de software (PRESSMAN, Roger, 2005), (SOMMERVILLE, Ian, 2005) se aborda una introducción a los métodos formales utilizando Z; además, también existen en el mercado gran variedad de libros y manuales especializados, por ejemplo (JACKY, Jonathan, 2008), (DILLER, Antoni, 1999), (BARDEN, Rosalind et al., 1994).

Por otro lado, las especificaciones de requisitos, formales o no, pueden ser reutilizadas, lo que implica aprovechar la experiencia y esfuerzo ganados en proyectos previos y aplicarlos adecuadamente en uno o más proyectos nuevos. Cualquier producto generado durante el ciclo de vida puede ser aprovechado nuevamente. De esta manera, desde las especificaciones de requisitos hasta el código y las pruebas son susceptibles de ser reutilizadas.

En los últimos años, la reutilización de software ha sido un medio eficaz para que los sistemas no sean desarrollados aisladamente unos de otros, ni se invierta esfuerzo en ellos comenzando su construcción siempre a partir de cero (NIU, Nan and Easterbrook, Steve, 2008). En un escenario con reutilización, los elementos se pueden enviar y extraer de repositorios donde se acumula, convenientemente clasificada, la experiencia obtenida.

Los elementos reutilizables almacenados y clasificados en los repositorios, deben ser correctos, coherentes, bien estructurados, con una buena modularización y una adecuada separación de intereses. De esta manera se evita la incorporación de elementos que degraden la calidad de los nuevos proyectos.

En este capítulo se presenta una introducción general al trabajo de tesis. La motivación para desarrollarlo se explica en el apartado 1.2. En el apartado 1.3 se presentan los objetivos del trabajo. Finalmente en el apartado 1.4 se muestra la organización en los apartados subsecuentes.

1.2. Motivación

Los temas de ingeniería de requisitos, aspectos tempranos, separación multidimensional de intereses, especificaciones formales y reutilización, de manera individual han sido objeto de diversos trabajos de investigación en los últimos años. Por ejemplo, se han propuesto modelos para realizar especificaciones incorporando aspectos (RASHID, Awais et al., 2002), (COLYER, Adrian et al., 2004), para incorporar un modelo multidimensional no formal en la ingeniería de requisitos (MOREIRA, Ana et al., 2005), para reutilizar especificaciones aspectuales (PARKASH, Kamal and Bhatia, Rajesh, 2007), para considerar métodos formales en las especificaciones aspectuales (NAFEES QAMAR, N. et al., 2006) y se han realizado extensiones de lenguajes formales existentes para incorporar aspectos (MOUSAVI, Mohammad et al., 2002) y (UBAYASHI, Naoyasu and Nakajima, Shin, 2007). Sin embargo, hay pocas aportaciones que reúnan todos los temas en un solo enfoque conjunto.

Por ejemplo, en (MOREIRA, Ana et al., 2005), se aborda la especificación de requerimientos con un enfoque multidimensional, pero sin introducir notaciones formales. También se propone el uso de un meta-espacio que contiene los conceptos de intereses más comunes para organizar los requerimientos capturados, pero sin una tendencia a la reutilización explícita de elementos en repositorios comunes a los proyectos de especificación. En este trabajo también se toma como base el modelo abstracto propuesto por Tarr y Sutton en (TARR, Peri and Sutton, Jr., Stanley M., 1999), y se restringen los tipos de proyecciones que los intereses pueden tener entre sí desde una perspectiva del lenguaje natural.

Por otra parte, no existe mucho trabajo significativo en notaciones formales que incorporen la orientación a aspectos en la Ingeniería de Requisitos utilizando lenguaje Z; por ejemplo, en la literatura, se propone un marco de trabajo para especificación y verificación formal de programas orientados a aspectos utilizando principalmente Object-Z (que es una extensión de Z tradicional para incorporar orientación a objetos) en (NAFEES QAMAR, N. et al., 2006); y en (YU, Huiqun et al., 2005) se expone un lenguaje llamado Aspect Z que también aumenta al lenguaje Z para incorporar aspectos. Ambos trabajos utilizan un enfoque bidimensional de separación de intereses, y no hacen aportaciones particulares en cuanto a la reutilización de especificaciones.

El lenguaje denominado Aspect Z fue propuesto como una extensión para incorporar aspectos al lenguaje Z. Aspect Z no utiliza construcciones especiales para especificar reglas de composición, sino esquemas tradicionales para encapsular los intereses transversales, donde se encuentran auto contenidos los datos necesarios para realizar la integración con el modelo base; éste último se especifica separadamente en otros esquemas.

El lenguaje agrega una notación para describir las relaciones de composición en los aspectos. Se deben crear dos modelos: uno base que contiene la especificación principal en Z y otro que utiliza Z con la extensión propuesta conteniendo los aspectos. Después del proceso de composición, se obtiene un modelo integrado que resulta de los aspectos atravesando a los requerimientos base.

Aspect Z posee una granularidad tal que permite realizar inserciones y reemplazos en la especificación de un esquema a nivel de puntos de enlace. Para definir el impacto de los aspectos en el modelo base se depende del contenido particular en el modelo base. Esto crea una dependencia que dificulta la reutilización del aspecto y el mantenimiento de las especificaciones.

Por ejemplo, considérese el siguiente ejemplo (YU, Huiqun et al., 2005) donde se tiene un modelo base de la siguiente manera:

ABaseModel

$a? : A$

$result! : B$

$f : A \longrightarrow B$

$result! = f(a?)$

y un modelo de aspectos, relacionado con el modelo base anterior de la siguiente manera:

AnAspectModel

$\Omega ABaseModel$

$Pointcut\ PC : \{f : A \longrightarrow B\}$

---

insert PC : $\forall x,y \in A \bullet x \neq y \longrightarrow *(x) \neq *(y)$

En el esquema *AnAspectModel* se indica insertar un predicado relacionado con *f* en el esquema *ABaseModel*. Se utiliza el símbolo Ω para denotar que *AnAspectModel* atraviesa *ABaseModel*. Nótense las referencias a la especificación base dentro del modelo aspectual. Este modelo de trabajo es influenciado fuertemente por el lenguaje de programación Aspect-J(LADDAD, Ramnivas, 2003). El modelo compuesto *ABaseModel* + *AnAspectModel* sería:

AnIntegratedModel

$a? : A$

$result! : B$

$f : A \longrightarrow B$

$\forall x,y \in A \bullet x \neq y \longrightarrow f(x) \neq f(y)$

$result! = f(a?)$

Si bien el modelo bidimensional que utiliza Aspect Z ha funcionado en la etapa de programación con algunos lenguajes como Aspect-J, tiene reconocidas limitaciones; por ejemplo, los aspectos mantienen una estrecha relación con los intereses base, es por eso que deben mantener un grado de compatibilidad implícito y deben conocerlos con suficiente detalle para poder actuar sobre ellos. Los aspectos además, no interactúan entre sí.

De esta manera se identificó un espacio de estudio factible en el área de la especificación formal de requerimientos con lenguaje Z para trabajar con una separación de intereses más enriquecida que la que ofrecía Z tradicional y con un soporte para aspectos más modular que el de Aspect Z; considerando además un medio para reutilizar especificaciones.

Para la propuesta de especificación formal de este trabajo se eligió lenguaje Z pues es una notación madura que ha sido utilizada en muchos proyectos y cuenta con una variada bibliografía en el mercado; además, Z se convirtió en estándar internacional (ISO/IEC 13568) en el año 2002. Por otra parte, se eligió adoptar una influencia multidimensional en la especificación de requerimientos por el alto nivel de separación de intereses que se logra con ese modelo.

En el Lenguaje Z tradicional, las especificaciones de requerimientos se realizan en estructuras denominadas esquemas que pueden relacionarse entre sí. Sin embargo, a pesar de que el lenguaje provee mecanismos para combinar especificaciones en esquemas separados, que bien pueden aprovecharse utilizando un enfoque orientado a aspectos, no cuenta con el concepto de regla de composición. En Z, para representar una combinación de dos esquemas se utiliza el cálculo de esquemas explicado en el apartado 14.17. Utilizar los mecanismos tradicionales de lenguaje Z implica especificar la inclusión de unos esquemas en otros. De esta manera, la separación no resulta completa ni totalmente organizada, porque se obtienen especificaciones anidadas que resultan difíciles de mantener especialmente si el número de esquemas involucrado es grande. Además, cuando se desea analizar el impacto de un esquema sobre otros, invariablemente se deben analizar cada uno de los esquemas "incrustados" involucrados en la especificación, haciendo el proceso más difícil aún.

Pensar separadamente en requerimientos (con su representación formal en esquemas) y en reglas de composición permite una organización de elementos más modular. Las reglas de composición contienen en sí mismas toda la información necesaria para que un requerimiento afecte a otros, de esta manera, basta modificar solamente la descripción de la regla para obtener los efectos deseados en la composición final sin alterar la estructura individual de los requerimientos involucrados. Una regla de composición es una construcción que agrupa un conjunto de acciones para efectuarse sobre los requerimientos y sus representaciones formales, por lo que también automatiza varias operaciones que no estarían disponibles directamente desde un esquema tradicional de Lenguaje Z.

1.3. Objetivos y contribuciones del presente trabajo

El objetivo de esta tesis es definir un modelo de trabajo para especificar requerimientos formales utilizando Lenguaje Z, con una separación de intereses simétrica que permita la correcta y homogénea especificación de intereses transversales y no transversales, y soportando la reutilización de los elementos involucrados en las especificaciones de una forma independiente y modular. El modelo de trabajo propuesto deberá permitir la futura evolución del propio modelo y la de los productos de la especificación a etapas posteriores del ciclo de vida de desarrollo.

A continuación se enumeran y explican las contribuciones del presente trabajo:

- Ampliación del lenguaje Z.- El presente trabajo hace uso del Lenguaje Z tal como ha sido definido por sus creadores; no impone restricciones sobre él, por el contrario, lo aumenta introduciendo los conceptos de requerimientos, intereses, reglas de composición y proyectos. Este modelo extendido se denomina Z-SIIMD (Z con Separación de Intereses con Influencia Multi-Dimensional).

- Reglas y composición.- Se propone también un mecanismo de relación que permita representar la proyección de unos intereses sobre otros, y un proceso de integración para obtener un modelo completo guiado por las relaciones establecidas. Las reglas contienen distintos elementos que permiten configurar el impacto de los requerimientos. El proceso de integración interpreta estos elementos y actúa en consecuencia produciendo un modelo final.

- Reutilización.- Se proponen repositorios para almacenar elementos reutilizables de los proyectos de especificación. El acceso a los repositorios está disponible mediante el proceso de reutilización, el cual incorpora las fases de clasificación y búsqueda e instanciación de elementos.

- Especificación formal con influencia multidimensional.- Para la separación de intereses se aplica una de las ideas principales del modelo genérico multidimensional propuesto por Tarr en (TARR, Peri and Sutton, Jr., Stanley M., 1999): una separación de intereses simétrica que no exija una descomposición inicial en dos modelos: uno base y otro aspectual, sino que trate homogéneamente a todos los intereses. Esta influencia permite a las especificaciones de Z-SIIMD contar con más oportunidades de composición y reutilización en distintos escenarios, al mismo tiempo que permite vislumbrar una evolución factible en el actual modelo de Z-SIIMD.
  En (TARR, Peri and Sutton, Jr., Stanley M., 1999), los autores también explican que las instanciaciones del modelo abstracto multidimensional pueden ser distintas en terminología y funcionalidad; como ejemplo, se comparan las características de varios enfoques existentes y se contrastan con el modelo abstracto.
  El presente trabajo, aunque se centra en la etapa de especificación de requerimientos, y no incorpora la terminología exacta del modelo abstracto multidimensional puro, representa un punto de partida para la inclusión de nuevas características que pueden ser incorporadas sobre el actual modelo, tomando como referencia la filosofía del modelo multidimensional puro. Por ejemplo, la capacidad de trabajar con varias descomposiciones de interés simultáneas en un mismo proyecto para maximizar la flexibilidad en la configuración de sus características, y la posibilidad de incluir más actividades del ciclo de vida del software relacionando los productos obtenidos entre ellas con un adecuado control de trazabilidad.

- Herramienta de software.- Aunque el objetivo principal de esta tesis no es la obtención de una herramienta computacional, se identificó la importancia de elaborar un prototipo (denominado SPZS, Software Prototipo para Z-SIIMD) que representara una posible implementación del enfoque propuesto en este trabajo. Entre otras cosas, el prototipo valida el algoritmo de composición, clarifica los mecanismos de clasificación e instanciación y corrobora la viabilidad de las propuestas realizadas.
  La primera versión de SPZS permite la captura de requerimientos, intereses y reglas de composición mediante una interfaz de usuario, y proporciona acceso al modelo compuesto con opciones para adecuar la selección de requerimientos que serán incluidos en él. También pone a la disposición de cada nuevo proyecto los repositorios con elementos reutilizables.

1.4. Organización

El contenido de esta tesis se organiza en tres partes principales. En la Parte I se encuentra el estado del arte de las áreas de la Ingeniería del Software que convergen en este trabajo; en la Parte II se aborda el desarrollo del trabajo de tesis y en la Parte III se encuentran las conclusiones y las líneas de trabajo futuro.

En el estado del arte (Parte I) se incluyen los siguientes temas: el Capítulo 2 aborda los métodos formales en la Ingeniería del Software; en el Capítulo 3 se estudia la reutilización de elementos de software; en el Capítulo 4 se explican dos enfoques relevantes para la separación de intereses y la orientación a aspectos, y finalmente en el Capítulo 5 se estudian los aspectos tempranos en el ciclo de vida del software.

En el desarrollo del presente trabajo (Parte II) se incluyen los siguientes capítulos: El Capítulo 6 contiene la presentación y los detalles generales de esta tesis; en el Capítulo 7 se muestra la descripción de los elementos del modelo de trabajo de Z-SIIMD. En el Capítulo 8 se aborda el modelo de composición. El Capítulo 9 explica el modelo de reutilización. En el Capítulo 10 se exponen las características de SPZS, el software prototipo elaborado para esta tesis. El Capítulo 11 muestra la aplicación del proceso de composición de especificaciones mediante un caso de uso para un agrupamiento (*pool*) de conexiones. Finalmente en el Capítulo 12 se muestra una aplicación de los procesos de reutilización e instanciación con un caso de uso para un retiro bancario.

En la Parte III, dentro del Capítulo 13, se encuentran las conclusiones y líneas de trabajo futuro de esta tesis. Finalmente, se incluye un apéndice para explicar las bases del Lenguaje Z en el Capítulo 14.

Parte I. Estado del arte

Capítulo 2.
Métodos Formales

2.1. ¿Qué son los Métodos Formales?

Un método formal es una técnica matemática para describir un sistema (BOWEN, Jonathan and Hinchey, Michael, 1995) que proporciona marcos de trabajo dentro de los que es posible especificar, desarrollar y verificar sistemas de manera metodológica (WING, Jeannette, 1990).

Un método se considera formal si: "(…) *tiene solidas bases matemáticas, usualmente proporcionadas por un lenguaje formal de especificación. Esta base ofrece los medios de definir con precisión nociones como consistencia y completitud y, con más relevancia, especificación, implementación y corrección*".(PRESSMAN, Roger, 2005).

En teoría, un método formal puede tener o no la ayuda de una herramienta de software; sin embargo, el análisis automático que éstas proporcionan es una de las grandes ventajas de trabajar con las especificaciones formales, por lo que siempre es preferible contar por lo menos con una que brinde soporte adecuado a cada metodología.

Los lenguajes de especificación formal proporcionan los medios para utilizar las bases matemáticas del método, las cuales consisten en: una notación particular para realizar las especificaciones (conocida como dominio sintáctico), un universo de objetos para definir el sistema (conocido como dominio semántico) y un conjunto de reglas que definen cuales objetos satisfacen cada especificación (denominadas relaciones).

2.2. Origen

Los métodos formales surgieron a finales de la década de los sesenta como una respuesta a la crisis del software (BANACH, Richard, 2007), que identificó la complejidad como la causante del fracaso de los proyectos de software a gran escala.

Como una manera de enfrentar los efectos de la complejidad y de asegurar más precisión en los proyectos desarrollados, se propuso un cambio en la manera de construir sistemas: un enfoque más formal en lugar del desarrollo aislado, personal y empírico que había prevalecido hasta entonces. También se acuñó la idea de que las matemáticas deberían tener un lugar especial en la ingeniería del software, tal como lo tienen en cualquier otra rama ingenieril.

De esa manera comenzaron diferentes investigaciones dando como resultado las primeras técnicas formales, las cuales constituyen las raíces de los actuales métodos.

2.3. Ventajas y desventajas

El uso de los métodos formales tiene marcadas ventajas que proporcionan razones importantes para utilizarlos en el desarrollo de software (HALL, Anthony, 1990), (SPIVEY, J. M., 1992), (BARDEN, Rosalind et al., 1994), (VIENNEAU, R., 1996). Algunas de ellas se enumeran a continuación:

- Mejoran la productividad en las actividades de desarrollo.
- Mejoran la calidad del software.
- Permiten una mayor comprensión del sistema.
- Mejoran la comunicación del equipo de desarrollo con el cliente, al contar con una especificación de requisitos sin ambigüedad.
- Al introducir rigor en el modelado y análisis, se puede describir al sistema de una manera más precisa.
- Es posible asegurar matemáticamente que el sistema es correcto de acuerdo a las especificaciones.
- Permiten verificar y validar la implementación.
- Permiten la detección temprana de errores.
- Los sistemas producidos son más fiables y seguros.
- Evitan gastos y problemas habituales en los métodos no formales.
- Reducen el riesgo.
- Facilitan el cumplimiento de regulaciones y estándares relacionados con la seguridad.

Por otra parte, el uso de métodos formales tiene ciertas limitantes o desventajas. Algunas de ellas han sido identificadas en (LAMSWEERDE, Axel Van, 2000), entre las que se mencionan:

- Su uso adecuado implica una importante curva de aprendizaje.
- Para que sea útil, un sistema desarrollado con métodos formales debe tener un dominio limitado de aplicación.
- Las especificaciones formales son difíciles de desarrollar. Difícilmente una especificación formal es correcta la primera vez que se realiza, aunque aun así, resulta útil.
- En los desarrollos ordinarios, que son mucho más comunes que los proyectos críticos, el interés hacia la calidad, seguridad y fiabilidad es escaso por lo que no se aplica formalidad comúnmente en ellos.
- No hay suficiente experiencia de aplicación de métodos formales en desarrollos ordinarios.

- Mucha de la experiencia en métodos formales ha surgido en la academia pero no se aplica inmediatamente en la industria debido a la reconocida brecha de transferencia tecnológica entre ambos campos.
- Las herramientas no son lo suficientemente populares, unificadas, maduras y amigables.
- Las herramientas de los distintos métodos no son compatibles entre sí.
- Los métodos formales tienen poca escalabilidad.

No ayudan a la popularización de los métodos formales diversos mitos que se han venido forjando con los años. En (HALL, Anthony, 1990) se mencionan los siete grandes mitos siguientes. Se asume que los métodos formales:

1. Pueden garantizar que el software es perfecto.- A pesar de que facilitan encontrar los errores, algunos detalles no siempre pueden ser probados, o las pruebas pueden fallar. Además, la especificación obtenida no siempre es perfecta.
2. Se refieren solo a "probar" la correctitud de los programas.- La prueba es solo una parte de la aplicación de los métodos formales. Otros usos comprenden modelado, comunicación y demostración.
3. Son útiles solo en sistemas de seguridad crítica.- En realidad lo son en cualquier sistema; aunque si se trata de un sistema crítico, éste debería ser obligadamente desarrollado utilizando métodos formales.
4. Requieren personal matemático altamente entrenado.- Las matemáticas para especificación requieren conocimientos básicos acerca de lógica y conjuntos; aunque si se desarrolla un proyecto crítico, siempre se debe contar con asesoría de expertos matemáticos.
5. Aumentan los costes de desarrollo.- En realidad los costes disminuyen, pero se debe invertir más tiempo en la etapa de especificación de requisitos.
6. No son aceptados por los usuarios.- Los usuarios reconocen la utilidad de los métodos formales si se les presentan adecuadamente; de esta manera, los usuarios entienden mejor la funcionalidad que el sistema les proporciona.
7. No son utilizados en proyectos de software reales y a gran escala.- Lo cierto es que están siendo utilizados en muchas áreas de la industria día con día. Las aplicaciones comprenden más que proyectos de seguridad crítica.

En (BOWEN, Jonathan and Hinchey, Michael, 1995) se retoman los mitos alrededor de los métodos formales y se reflexiona sobre otros siete más:

1. El uso de métodos formales retrasa el proceso de desarrollo.- Si bien es cierto que muchos proyectos desarrollados con métodos formales se han ralentizado, la verdadera causa ha sido la mala estimación de recursos y no la inclusión de la formalidad en el proceso.
2. Hay una carencia clara de herramientas de soporte para el uso de métodos formales.- La tendencia es que los métodos formales tengan sus respectivas herramientas asociadas. Hoy en día, esto se ha reconocido como una manera de aumentar la productividad en desarrollos formales.

3. Los métodos formales conseguirán reemplazar a los métodos de diseño tradicionales.- En los últimos años, ambos tienden a integrarse para complementar sus respectivas carencias.
4. Los métodos formales se aplican únicamente al software.- En realidad se han aplicado exitosamente tanto en software como en hardware.
5. No es preciso aplicar metodologías formales.- Los métodos formales son imprescindibles, además de en los sistemas críticos, en cualquier sistema en el que la corrección sea importante.
6. No existe el conocimiento de base para la implantación de las metodologías formales.- Actualmente el interés por los métodos formales ha crecido: se han incluido materias que los abordan durante las carreras profesionales y se han formado múltiples foros de consulta.
7. Siempre se deben aplicar metodologías formales para el desarrollo de sistemas software.- Existen casos como el diseño de las interfaces de usuario en donde los métodos formales no encajan completamente y es mejor utilizar un enfoque tradicional menos riguroso.

Hoy en día los métodos formales pueden ser vistos todavía como imprácticos o exclusivos de un trabajo académico-teórico. Sin embargo, se reconocen bien sus beneficios, aunque son utilizados principalmente cuando el sistema a desarrollar es pequeño o cuando se tiene una necesidad de alto cumplimiento de los requerimientos (BISHOP, Matt et al., 2011). En la última década, los métodos formales y sus herramientas de software asociadas han mostrado una tendencia a incorporar la semántica formal de algunos lenguajes de programación modernos, logrando con esto acelerar la transición del terreno académico al comercial (SCHAEFER, Ina and Hähnle, Reiner, 2011).

2.4. Taxonomía de las especificaciones formales

Las especificaciones formales se pueden realizar desde diferentes perspectivas, las cuales son dictadas por los diferentes paradigmas de especificación formal. Cada uno de ellos se diferencia del resto por la orientación principal considerada para elaborar las especificaciones. En (LAMSWEERDE, Axel Van, 2000) y (LAMSWEERDE, Axel Van, 2001) se propone la siguiente taxonomía: especificaciones orientadas a comportamientos, a estados, a transiciones, funcionales y operacionales.

2.4.1. Especificación orientada a comportamientos

Este paradigma permite especificar un sistema mediante el conjunto máximo de sus conductas admisibles a lo largo del tiempo. La especificación de propiedades relevantes se realiza mediante expresiones de lógica temporal sobre los objetos del sistema. La lógica temporal utilizada permite incorporar operadores relacionados con el pasado, el presente y

el futuro. Las expresiones tienen significado sobre ciertas estructuras de tiempo, las cuales pueden ser de naturaleza discreta, densa o continua. Las propiedades también pueden hacer referencia a puntos específicos en el tiempo, intervalos, o ambos.

Las referencias de tiempo incluidas en las lógicas temporales permiten que no sea necesario introducir variables adicionales para representar el pasado o el futuro, con lo que se simplifica el proceso de especificación.

Se ha identificado a la especificación orientada a comportamientos como la más apropiada para las etapas tempranas de la especificación de requerimientos. Todas las variantes de la lógica temporal son ejemplos de lenguajes de esta categoría.

2.4.2. Especificación orientada a estados

Mediante este paradigma, es posible describir un sistema mediante sus estados admisibles en un punto arbitrario de tiempo. La especificación de propiedades de interés se realiza considerando un instante en el tiempo, utilizando invariantes que restringen los objetos del sistema y precondiciones y post-condiciones que restringen la ejecución de sus operaciones.

Una precondición engloba la condición más débil que es necesaria en los estados de entrada para que la operación pueda realizarse. Por otro lado, una post-condición define la condición más estricta que debe observarse sobre los estados de salida cuando se realiza la operación.

La especificación orientada a estados resulta más operativa que la orientada a comportamientos, por lo que suele ser más apropiada para las últimas etapas de la especificación de requerimientos. Ejemplos de lenguajes de esta categoría son Z (SPIVEY, J. M., 1992), VDM (HAYES, I. J. et al., 1994) y B (ABRIAL, J. R., 1996).

2.4.3. Especificación orientada a transiciones

Este paradigma permite describir un sistema mediante transiciones entre estados. La especificación de propiedades de interés se realiza usando funciones de transición en máquinas de estados. Cada función de transición define el estado de salida correspondiente para cada estado de entrada y evento. El hecho de que ocurra un evento es condición suficiente para que se lleve a cabo la transición. También se pueden especificar precondiciones adicionales como guardas. Ejemplos de lenguajes de esta categoría son Statecharts (HAREL, David, 1986) y PROMELA (MORDECHAI, Ben-Ari, 2008).

2.4.4. Especificación funcional

Este paradigma permite describir un sistema como un conjunto de funciones matemáticas. Se pueden distinguir dos enfoques: la especificación algebraica y la especificación con funciones de orden superior.

2.4.4.1. Especificación algebraica

En la especificación algebraica, las funciones se agrupan por tipos de datos abstractos, los cuales son definiciones de tipos de datos que incluyen operaciones para manejarlos. Lenguajes típicos de esta categoría son LARCH (GUTTAG, John et al., 1985) y la familia de lenguajes OBJ (GOGUEN, Joseph and Grant, Malcom, 1996) .

2.4.4.2. Especificación con funciones de orden superior

En la especificación con funciones de orden superior, las funciones se agrupan en teorías lógicas. Las teorías contienen definiciones de tipos, declaraciones de variables y axiomas. Las funciones también pueden tener otras funciones como argumentos, lo que resulta en lenguajes más poderosos. Algunos lenguajes de este paradigma son HOL (NIPKOW, Tobias et al., 2002) y PVS (OWRE, Sam et al., 1992).

2.4.5. Especificación operacional

La especificación operacional permite modelar un sistema como una colección estructurada de procesos concurrentes que puede ser ejecutada por una máquina más o menos abstracta. Este paradigma se encuentra muy apegado al nivel de programación. Ejemplos de lenguajes que soportan este paradigma son las redes Petri (AGERWALA, Tilak, 1979) y las algebras de procesos (FOKKINK, Wan, 2000).

2.5. Los métodos formales en las diferentes actividades del ciclo de vida

Un proceso de desarrollo de software, también conocido como ciclo de vida, es una serie de pasos que involucran actividades, restricciones y recursos para describir la vida de un producto de software desde su concepción hasta su implementación, entrega, utilización y mantenimiento (LAWRENCE, Pfleeger, 2002).

Existen varios modelos del proceso de desarrollo de software. Cada uno enumera y ordena las actividades para desarrollar un sistema de software de diferente manera. Cada modelo

encapsula estas actividades en fases específicas que tienen un objetivo claro. Cada fase recibe entradas y produce salidas; estas últimas sirven como punto de partida (entradas) en la siguiente fase.

La mayoría de los modelos tienen como común denominador una serie de fases generales, las cuales se enriquecen, adecúan y relacionan entre sí de acuerdo a cada modelo en particular. Estas fases son: análisis de requerimientos, diseño, implementación, prueba, operación y mantenimiento.

Los métodos formales pueden aplicarse en todas estas fases, como parte integral del proceso de desarrollo y no de forma separada (WING, Jeannette, 1990).

En la fase de análisis de requerimientos se extraen, refinan y especifican las necesidades de los usuarios que deben ser cumplidas por el sistema a desarrollar. En esta fase, los métodos formales ayudan a clarificar los requerimientos que el cliente expresa de manera informal; además, son útiles para revelar contradicciones, ambigüedades e incompletitudes. El resultado es una especificación formal, la cual se considera también una documentación del sistema, ya que sirve como comunicación entre el cliente, el especificador y el implementador.

La fase de diseño determina la arquitectura del sistema necesaria para satisfacer los requerimientos especificados. Primero se realiza un diseño de alto nivel de los módulos y posteriormente se lleva a cabo un diseño detallado que debe considerar un suficiente grado de descomposición para llevarse a la fase de implementación.

En la fase de diseño, se puede hablar de la participación de los métodos formales en las tareas de descomposición y el refinamiento. La descomposición consiste en dividir el sistema en módulos más pequeños. El refinamiento implica trabajar correctamente con el sistema en diferentes niveles de abstracción; por ejemplo, suponer un nivel muy general y varios niveles más específicos que le agregan detalles.

Para la descomposición, se escriben especificaciones formales con el objetivo de definir las interfaces entre los módulos. En el refinamiento, mediante métodos formales es posible demostrar que la especificación de un nivel determinado satisface también la especificación de su nivel superior.

En la fase de implementación, se realiza la construcción del sistema, persiguiendo su correcta ejecución en el equipo y la plataforma seleccionados. Si se ha llevado un proceso formal de desarrollo hasta este punto, es posible utilizar herramientas computacionales que generen código de manera automática o semi-automática a partir de las especificaciones formales de las etapas anteriores. Con esto se garantiza que el código obtenido se apega completamente a los requerimientos especificados inicialmente.

En la fase de prueba se tiene como meta asegurarse que la implementación satisface los requisitos especificados (verificación), así como las expectativas del cliente (validación). A estas dos actividades en conjunto se les denomina “proceso de verificación y validación”.

La verificación asegura que los productos estén completos y sean consistentes con otros y con las necesidades del cliente (FAIRLEY, Richard, 1994). De esta manera, por ejemplo, se puede verificar que el diseño esté completo y corresponda con la especificación de requerimientos, o también se puede verificar que el código fuente corresponda con el diseño y con los requerimientos. En esta etapa, los métodos formales ayudan a demostrar que un sistema satisface los requerimientos establecidos en su especificación formal mediante rigurosas demostraciones matemáticas. Esto puede llevarse a cabo en todo el sistema, o solo en ciertas partes seleccionadas estratégicamente.

La validación se relaciona con probar la calidad del software en su medio de operación, e involucra la planeación y realización de casos prueba. La validación consiste en la evaluación del software al final del proceso de desarrollo para determinar su conformidad con los requerimientos (FAIRLEY, Richard, 1994). En esta etapa, los métodos formales son útiles durante las pruebas y la depuración. Las especificaciones formales pueden usarse para producir casos de prueba de caja negra o de condiciones límite; además, si se combinan con la implementación, las especificaciones también pueden usarse para pruebas de rutas, de unidad e integración.

En la fase de operación y mantenimiento, el sistema completo y funcional se instala para que el cliente lo utilice. De esta manera surge la necesidad de llevar a cabo modificaciones ya sea de corrección, de adaptación o de mejora en las características ofrecidas por el software. En esta fase, las especificaciones formales ayudan a realizar una evaluación crítica del sistema, sirviendo como punto de referencia para analizar el desempeño del software construido. En esta etapa se pueden realizar ajustes a las especificaciones y al sistema mismo para que concuerden con las situaciones reales del entorno operacional.

2.5.1. Los métodos formales en la especificación de requerimientos

Durante la etapa de análisis y especificación de requerimientos se estudian, entienden y ordenan las necesidades que el usuario espera satisfacer con el nuevo sistema. El objetivo que se persigue en esta fase es capturar estas necesidades en un documento que servirá como guía para el desarrollo del proyecto.

Concretamente, un requerimiento es la descripción de una característica que un sistema es capaz de realizar para satisfacer su propósito global. El documento generado tras el análisis de necesidades se llama especificación de requerimientos.

La especificación de requerimientos es muy importante para asegurar que el sistema a desarrollar cumpla su cometido. Las decisiones que se tomen en el resto de las fases del ciclo de vida dependen en gran medida del contenido de esta especificación. Así que si ésta es vaga, incompleta, inconsistente, o simplemente no refleja las necesidades reales del cliente, el proyecto tenderá a fracasar.

Al aplicar un método formal en la etapa de especificación de requisitos se obtiene un documento detallado que sirve como medio de comunicación entre el cliente, el especificador y el implementador. Este documento contiene la especificación rigurosa y precisa de los requerimientos que el sistema a realizar debe satisfacer.

Una especificación formal contribuye a una buena captura de requerimientos, reduciendo por consecuencia la introducción de problemas posteriores en el proceso de desarrollo, que resultan en consumo adicional de tiempo, trabajo y presupuesto, o incluso pueden provocar el fracaso total del proyecto.

La participación del cliente es necesaria y muy importante al principio del proceso de especificación; en las etapas tempranas las tareas deben estar centradas en él. Sin embargo, esta tendencia cambia al ir agregando más detalle a las especificaciones del sistema; entonces el papel principal pasa a ser del equipo de desarrollo, quien se encarga de diseñar y especificar los detalles de implementación del sistema de acuerdo a las requerimientos iniciales. La Figura 2.1 muestra este escenario gráficamente.

Las etapas de especificación mostradas en la Figura 2.1 se pueden llevar a cabo en un orden diferente. Por ejemplo, en la Figura 2.2 se muestran las actividades de especificación que pueden realizarse paralelamente y la relación que existe entre ellas; es importante observar que existe una relación bidireccional entre las actividades de requerimientos y las de diseño, pues las primeras encaminan las segundas, y las segundas retroalimentan las primeras.

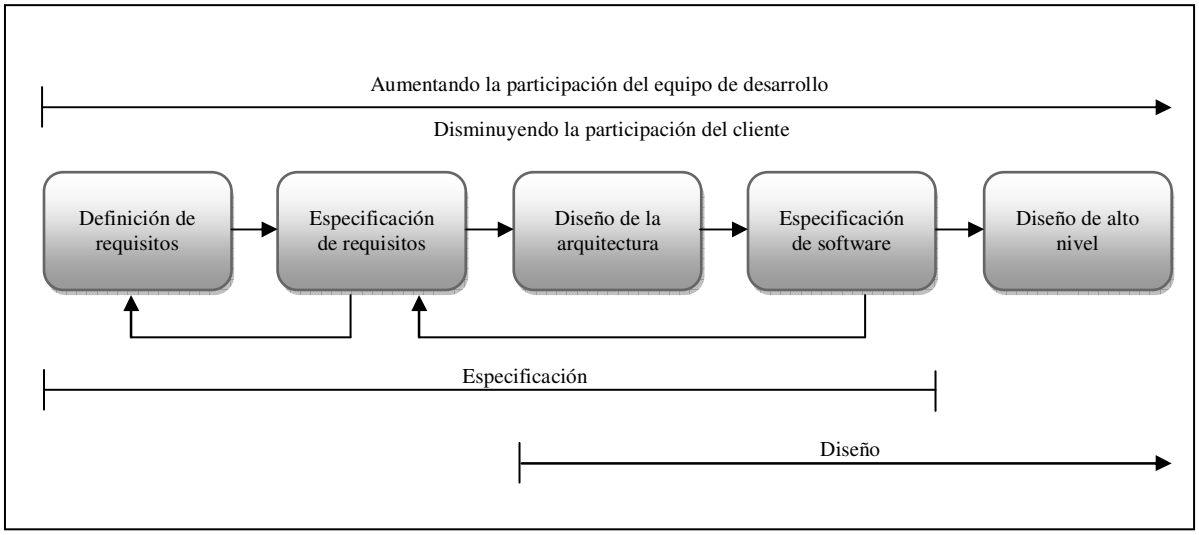


Figura 2.1. Especificación y diseño. (SOMMERVILLE, Ian, 2005).

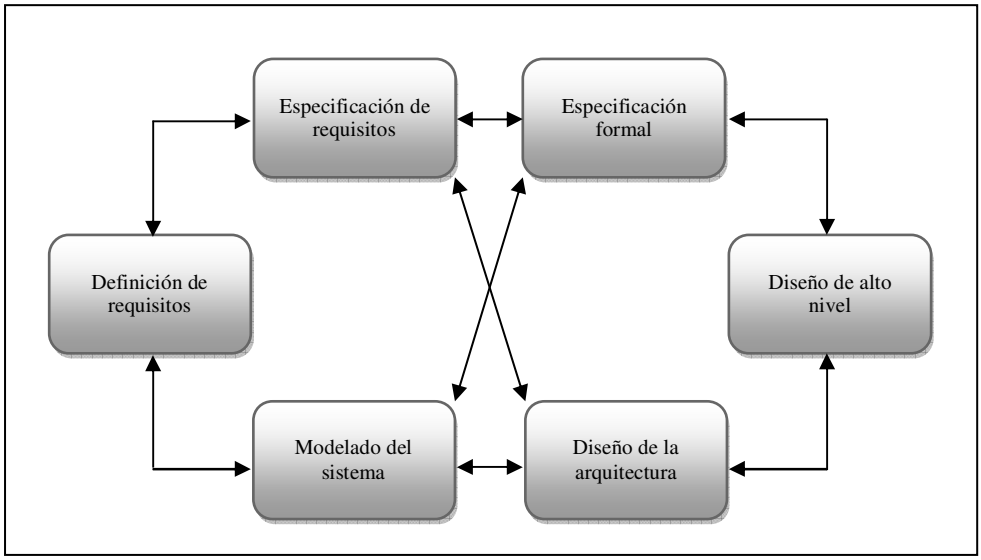


Figura 2.2. La especificación formal en el proceso del software. (SOMMERVILLE, Ian, 2005).

2.6. Enfoques para la especificación formal

La manera de realizar la especificación formal en el ciclo de vida de desarrollo de software ha evolucionado como resultado del aumento en la complejidad en el desarrollo de los sistemas en los últimos años.

Se puede hablar de tres enfoques distintos en la aplicación de la especificación formal; éstos son: enfoques unitarios, enfoques incrementales y enfoques ágiles.

Hace varias décadas, al inicio de la aplicación de los métodos formales, la naturaleza de los sistemas desarrollados permitía utilizar un enfoque unitario, donde las especificaciones se elaboraban completas en una sola etapa. Esta forma de trabajar resultaba apropiada y suficiente porque la complejidad de los sistemas no era tan grande como lo sería en años posteriores.

Tiempo después, surgió la necesidad de enfrentar la creciente complejidad con un enfoque incremental. El plan principal era lograr la descomposición de la complejidad y el refinamiento progresivo de las ideas. Con este planteamiento se puede partir de una idea inicial incompleta de la funcionalidad del sistema a desarrollar, para posteriormente refinarla mediante varias iteraciones. Cada iteración implica la adición de nuevas características al sistema o la descripción del sistema de manera más detallada.

Cada refinamiento debe validar el trabajo realizado en esa iteración para poder avanzar a la siguiente. De esta manera se van creando versiones parciales incompletas pero validadas,

que permiten que la siguiente versión se concentre en considerar nuevas características del sistema que no están presentes en la versión actual.

Una desventaja de este modelo radica en que si en algún refinamiento se descubre un problema, se asume que el último refinamiento completado es el causante del mismo. Por lo tanto, es imposible realizar cambios a refinamientos anteriores (ver Figura 2.3). Esta visión contradice la realidad actual del desarrollo de sistemas, en donde las necesidades no se pueden conocer completamente con anticipación y las especificaciones, incluso las definidas más tempranamente, deben actualizarse sobre la marcha.

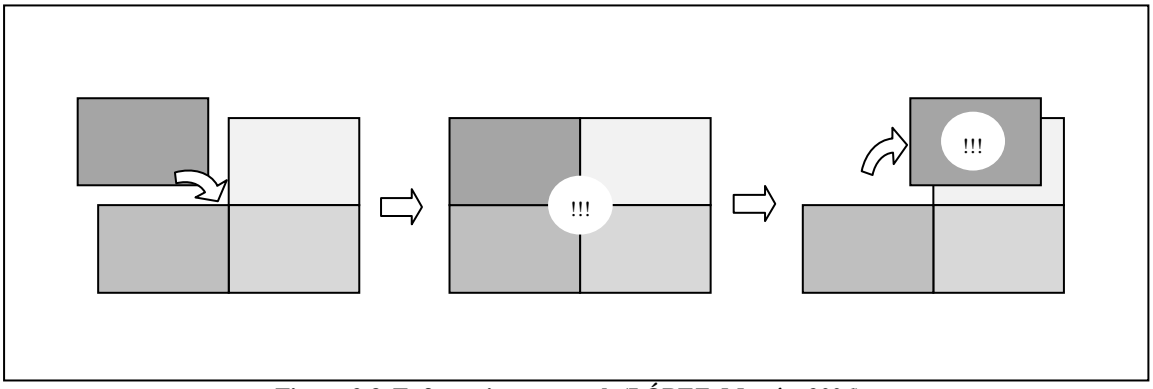


Figura 2.3. Enfoque incremental. (LÓPEZ, Martín, 2006).

Partiendo de este argumento nacen los enfoques ágiles (ver Figura 2.4), los cuales reconocen el cambio como algo constante en el proceso de desarrollo y proponen un esquema iterativo en el que:

- Es posible realizar modificaciones frecuentemente.
- Se reconoce al recurso humano como motor principal del éxito de un proyecto.
- Se promueve el trabajo creativo de los desarrolladores.
- Se promueve la participación de los usuarios finales.
- Se trabaja con el conocimiento que se tenga disponible, aunque no esté completo o no sea correcto en ese momento.
- Se promueve la re-modularización (*refactoring*) del sistema desde el principio y continuamente, al avanzar en el desarrollo, de manera independiente de la arquitectura.
- Si se encuentran problemas en las comprobaciones, se puede revisar cualquier parte del trabajo, no solo la última modificación como sucede en el enfoque incremental.

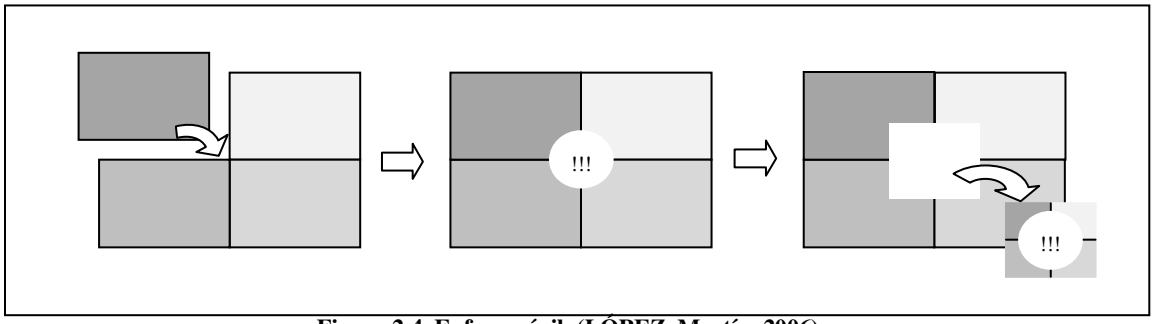


Figura 2.4. Enfoque ágil. (LÓPEZ, Martín, 2006).

Actualmente existe mucho interés en la investigación de los enfoques formales ágiles, por los resultados positivos que se han observado al aplicar los principios de agilidad en general en la ingeniería del software. Se persigue que los métodos formales agiles se conviertan en una solución sólida para enfrentar las especificaciones cambiantes; sin embargo, los enfoques formales ágiles actualmente están limitados en cuanto al soporte metodológico que ofrecen por las siguientes razones:

- Solamente incorporan algunas de las recomendaciones a nivel de código fuente que hacen principalmente la programación extrema (*eXtreme Programming*) y el Proceso Unificado (*Rational Unified Process*).
- Las herramientas de software existentes no proporcionan soporte adecuado para cambios frecuentes y numerosos.
- Las herramientas de software existentes no detectan ni informan las posibles causas de los problemas que surgen en las especificaciones.
- No hay soporte adecuado para la remodularización, y en algunos casos ni siquiera se dispone de mecanismos de modularización.

2.7. Descripción de algunos lenguajes formales

El presente apartado contiene una breve descripción de algunos de los lenguajes formales existentes más comunes. Se abordan: B, CCS, CSP, ESTELLE, HOL, LARCH, Lógica Temporal, LOTOS, OBJ, PROMELA, PVS, Redes Petri, SDL, STATECHARTS, VDM y Z. Al final se presentan unas reflexiones basadas en la comparación de sus principales características.

B

Su nombre completo es B-Technologies (ABRIAL, J. R., 1996). Consiste de tres componentes: el Método B (*B Method*), la herramienta B (*B tool*) y el paquete de herramientas B (*B toolkit*).

El método B es una notación y metodología para: modelado de requerimientos, especificación de interfaces, diseño, implementación y mantenimiento de software. El lenguaje utilizado es AMN (*Abstract Machine Notation*) que permite que las especificaciones sean validadas dinámicamente y verificadas matemáticamente. B Soporta la verificación y validación incremental, así como la construcción incremental del software.

El paquete de herramientas B consiste en un conjunto de herramientas que soportan el método B. Las herramientas pueden ejecutarse automáticamente o de manera interactiva desde un entorno gráfico.

CCS

CCS (*Calculus of Communicating Systems*) (MILNER, Robin, 1980) proporciona una manera de especificar sistemas concurrentes. En CSS se utilizan conjuntos de términos, operadores y axiomas que sirven para la definición y manipulación de expresiones, mediante las cuales se representa un sistema concurrente, con propiedades y comportamiento.

Con CSS se pueden modelar agentes secuenciales y alternativos; además es posible representarlos de manera gráfica si se utilizan herramientas como el CWB (*Concurrency WorkBench*). CWB puede simular un sistema concurrente previamente descrito en CCS, y puede detectar estados *deadlock*, realizar pruebas de igualdad y determinar si un sistema posee ciertas propiedades o no.

CSP

CSP (*Communicating Sequential Processes*) (DAVIES, Jim, 1993) proporciona un lenguaje para describir patrones de interacción. Fue creado por Hoare a finales de la década de los setenta, y su estructura tiene una gran influencia del lenguaje ADA.

En CSP la comunicación entre procesos se logra mediante el paso de mensajes con memoria compartida por canales bien definidos, y es posible describir de una manera muy natural los sistemas paralelos con múltiples entradas y salidas.

ESTELLE

Fue desarrollado a mediados de los ochenta, teniendo como principal propósito especificar protocolos de comunicación. Sin embargo, puede especificar sistemas distribuidos en general. En 1989, ESTELLE (INTERNATIONAL ORGANIZATION FOR STANDARIZATION, 1989) se aprobó como estándar internacional ISO. ESTELLE se considera un lenguaje expresivo, bien estructurado y definido para especificar sistemas de

procesamiento concurrente y distribuido. Existen varias herramientas que lo soportan, entre ellas *Estelle Development Toolset*.

HOL

HOL (*Higher Order Logic*) (NIPKOW, Tobias et al., 2002) es un sistema para probar teoremas en lógica de orden superior de manera interactiva. Su lógica formal se encuentra ligada a un lenguaje de propósito general llamado ML (*Meta-Language*), el cual sirve para escribir teoremas, y estrategias de prueba, así como para desarrollar teorías lógicas.

Inicialmente HOL fue creado para especificar y verificar diseños de hardware. Sin embargo, su lógica no aplica restricciones al respecto, y hasta la fecha ya ha sido aplicada exitosamente en diversas áreas en proyectos de hardware y software.

Existen tres versiones de HOL: HOL88, HOL90 y HOL98. La principal diferencia entre las tres es la variante de ML que incorporan. Funcionalmente no hay grandes cambios que las distingan.

LARCH

LARCH (GUTTAG, John et al., 1985) es una familia de lenguajes para especificación formal. Su idea inicial nació en la década de los setenta y su desarrollo ha sido apoyado por diferentes compañías como ARPA, NSF, Digital Equipment Corporation y Xerox.

Cada especificación realizada en LARCH contiene componentes escritos en dos lenguajes: uno, relacionado con un lenguaje de programación específico (*Larch Interface Languages* – LILs-) y otro independiente del lenguaje de programación (*Larch Shared Language* -LSL-).

Las interfaces entre los componentes del programa se especifican usando los LILs, que ayudan a expresar la manera de comunicación propia del lenguaje de programación utilizado. Se han diseñado lenguajes para: Ada, C, C++, CLU, CORBA, ML, Modula-3 y SmallTalk. También existen lenguajes con características genéricas que pueden extenderse a nuevos lenguajes de programación.

Por otra parte, los lenguajes LSL permiten definir nuevos términos para especificaciones de interfaces o clases especiales no incluidas en los LILs.

Lógica temporal

Las lógicas temporales (BELLINI, P. et al., 2000) se utilizan en las fases de análisis de requerimientos, análisis avanzado, especificación y ejecución. Su naturaleza les permite mantener un enfoque de modelado orientado más a la conducta del sistema y no tanto sobre los detalles funcionales o estructurales de los sistemas.

Las lógicas temporales típicamente introducen cuatro operadores de tiempo para extender las lógicas clásicas. El operador G significa "Siempre en el futuro", y es representado también por el símbolo □, el operador F quiere decir "Eventualmente en el futuro", y es denotado también por ◊, el operador H tiene el significado de "Siempre en el pasado", y está representado también por ■)y el operador P quiere decir "Eventualmente en el pasado", y es denotado también por ♦.

La mayoría de las lógicas temporales existentes han sido derivadas a partir de extensiones lógicas proposicionales o de primer orden, y raramente a partir de lógicas de orden superior (HOL –*High Order Logics*-).

LOTOS

LOTOS (*Language Of Temporal Ordering Specification*) (VAN EIJK, P. H. J. et al., 1989) es una notación formal que fue creada con la idea de expresar formalmente normas de servicios y protocolos OSI.

Las especificaciones formales se realizan en LOTOS con la visión de expresar las relaciones del sistema con su entorno, soslayando en lo posible la estructura interna del sistema. Otra característica importante de las especificaciones en LOTOS es que permiten mantener la arquitectura libre de conexiones con futuras implementaciones.

LOTOS está conformado por dos sub-lenguajes. Uno es un algebra de procesos que permite expresar comportamientos e interacciones entre procesos; este lenguaje está basado en los lenguajes CCS, CSP y CIRCAL. El segundo lenguaje se encarga de la descripción de tipos de datos abstractos y está basado en el lenguaje ACT-ONE; este lenguaje es independiente del anterior y puede ser sustituido por cualquier otro que proporcione una funcionalidad similar. Ambos lenguajes se complementan y permiten describir los sistemas de manera completa.

OBJ

OBJ (GOGUEN, Joseph and Grant, Malcom, 1996) es una familia de lenguajes de programación algebraicos de amplio espectro. Todos son lenguajes lógicos: sus programas consisten en conjuntos de sentencias lógicas y operan mediante deducciones en ellas.

Todos los lenguajes OBJ proporcionan mecanismos para utilizar subtipos, manejo de errores, herencia múltiple, y sobrecarga. También permiten el uso de módulos con parámetros. Ejemplos de algunos lenguajes de esta familia son: OBJ3, BOBJ, y CAFEOBJ.

En particular, OBJ3 ha sido utilizado exitosamente para investigación y enseñanza en las áreas de especificación y diseño de software, prototipado rápido, prueba de teoremas, diseño de interfaces de usuario y verificación de hardware.

PROMELA

PROMELA (*PROcess/PROtocol MEta-LAnguage)* (MORDECHAI, Ben-Ari, 2008) es un lenguaje formal que permite modelar sistemas distribuidos. Típicamente se utiliza en conjunto con SPIN, una herramienta de análisis que permite verificar la conducta de los procesos especificados en PROMELA. Ambos han sido muy útiles en el modelado y verificación de protocolos de comunicación.

La sintaxis de PROMELA es muy parecida al lenguaje C. Los programas están compuestos de procesos, canales de mensajes y variables. Los procesos permiten representar la conducta de los objetos concurrentes de los sistemas distribuidos. Los canales de mensajes y las variables definen el entorno de ejecución en el que se ejecutan los procesos.

SPIN proporciona los medios para realizar análisis de consistencia lógica en los modelos de PROMELA; permite realizar simulaciones de la ejecución del sistema y puede generar programas en C que verifican el espacio de estados del sistema.

PVS

PVS (*Prototype Verification System*) (OWRE, Sam et al., 1992) es un conjunto de elementos integrados en un entorno computacional con el objetivo de facilitar el desarrollo y análisis de especificaciones formales. Los elementos que integran a PVS son: un lenguaje de especificación, un numero de teorías predefinidas, un verificador de tipos, un probador de teoremas interactivo, varias utilerías como un generador de código, librerías y documentación.

El lenguaje de especificación de PVS utiliza lógica de orden superior y tiene integrado un nutrido conjunto de tipos, subtipos y constructores de tipos. Las especificaciones se realizan mediante el uso de definiciones, axiomas o ambas, y se organizan en teorías y tipos de datos parametrizados.

PVS se ha utilizado en áreas como análisis, teoría de grafos, teoría de números, verificación de hardware, y algoritmos secuenciales y distribuidos.

Redes Petri

Las redes Petri (AGERWALA, Tilak, 1979) fueron presentadas en 1962 por Carl Adam Petri en su tesis doctoral defendida en el seno de la Universidad Técnica de Darmstadt, Alemania.

Las redes Petri son una herramienta matemática y gráfica de modelado, útil para describir y analizar sistemas concurrentes, asíncronos, distribuidos, paralelos, y no determinísticos.

Por su naturaleza gráfica, las redes Petri sirven como medio de comunicación visual; por su naturaleza matemática, pueden ser relacionadas con ecuaciones de estados, ecuaciones algebraicas y otros modelos matemáticos que describan la conducta de los sistemas representados. Esta dualidad permite que las redes Petri sean un medio poderoso de comunicación entre los desarrolladores y los teóricos.

Las redes Petri han sido utilizadas exitosamente en áreas como la evaluación de desempeño de sistemas y los protocolos de comunicación.

SDL

SDL (*Specification and Description Language*) (ROCKSTRÖM, Anders and Saracco, Roberto, 1982) aparece en su versión inicial en 1972 con el apoyo de compañías como Ericsson y Motorola; posteriormente se fueron refinando las versiones hasta completar la actual, que es más completa y poderosa. En todo este tiempo, SDL ha sido muy utilizado en la industria, y han surgido múltiples herramientas que lo soportan.

SDL permite modelar sistemas con características temporales y de indeterminismo mediante componentes de: estructura, comunicación, comportamiento, datos y herencia. Un modelo básico descrito en SDL consiste en un conjunto de maquinas finitas (FSMs – *Finite State Machines*-) independientes intercomunicadas entre sí mediante señales discretas.

STATECHARTS

STATECHARTS (Diagramas de estados) (HAREL, David, 1986) fue presentado por Harel en la década de los ochenta como un medio gráfico de diseño que permitía especificar objetos. STATECHARTS es especialmente útil para representar los diversos estados por los que un sistema puede atravesar y las correspondientes transiciones entre esos estados.

A pesar de que STATECHARTS no ignora el aspecto dinámico de los sistemas, se considera que su principal modelo de trabajo se concentra en capturar aspectos estáticos de un sistema, ya que un estado es equivalente a una fotografía instantánea tomada al sistema en un momento dado. Una transición entre dos estados está relacionada con un evento, que

al ocurrir provoca el cambio de estado del sistema. Las transiciones opcionalmente incluyen guardas y acciones.

Los estados en STATECHARTS pueden ser simples o compuestos. Un estado simple consiste de dos estados y una transición entre ellos. Un estado compuesto es aquel que contiene sub-estados en su interior.

Un entorno computacional que utiliza STATECHARTS es StateMate, simultáneamente con un lenguaje orientado a bloques llamado ActivityCharts.

VDM

El término VDM es una abreviatura para "*Vienna Development Method*", un sucesor de los métodos gestados entre 1960 y 1970 en el laboratorio de IBM en Vienna. A la fecha ha sido utilizado exitosamente a nivel académico e industrial en el desarrollo de compiladores, bases de datos, sistemas de almacenamiento tolerantes a las fallas, software gráfico, sistemas médicos, etc.

Tres componentes son los que conforman a VDM (HAYES, I. J. et al., 1994): un lenguaje de especificación llamado VDM-SL, un conjunto de reglas para datos y refinamiento de operaciones que permiten relacionar las especificaciones abstractas de requerimientos con las especificaciones detalladas de diseño, y una rigurosa teoría de pruebas para verificación.

VDM-SL está basado en lógica de funciones parciales (LPF), en la que una expresión puede ser verdadera, falsa o indefinida. Contario a Z, las pre-condiciones y post-condiciones se deben escribir explícitamente.

Aunque la modularidad y la concurrencia son características que no están presentes en la versión estandarizada de VDM, hay trabajos de investigación y herramientas relacionadas que las incorporan.

Z

El lenguaje Z (SPIVEY, J. M., 1992) fue desarrollado en la Universidad de Oxford a principios de los ochenta; a la fecha es uno de los más utilizados y es soportado por una amplia gama de herramientas.

Z proporciona una notación formal que modela sistemas utilizando la teoría de conjuntos y la lógica de primer orden. Z ha ampliado los conceptos matemáticos convencionales y los ha enriquecido con constructores que permiten la especificación de software.

En una especificación se pueden introducir nuevos tipos mediante productos cartesianos, conjuntos o secuencias, o bien, tipos abstractos o por extensión.

En Z existen esquemas, los cuales contienen variables de estado, restricciones y operaciones sobre ese estado. También son posibles las operaciones de composición, renombrado y ocultado de esquemas. De manera implícita se pueden especificar pre-condiciones y post-condiciones.

Z modela adecuadamente y de manera natural las entidades estáticas con su respectivo comportamiento; no así algunos conceptos dinámicos como el tiempo real o la concurrencia. Sin embargo, al respecto existen trabajos de investigación que realizan propuestas para superar estas limitaciones.

2.8. Conclusiones

Los lenguajes formales presentados difieren entre sí fundamentalmente en la base matemática y en el modelo de trabajo que utilizan; sin embargo, comparten el mismo objetivo: todos proporcionan medios formales para que se puedan realizar especificaciones de los sistemas a desarrollar. Aunque cada lenguaje posee diferentes características para obtener el mismo resultado, el desarrollador debe conocer más acerca de cada uno de ellos cuando se trata de elegir el más adecuado para especificar un sistema.

El dominio del problema al cual pertenece el sistema a especificar es un factor importante para seleccionar un lenguaje particular. Así, por ejemplo, si se trabaja con un sistema en tiempo real se debería considerar un lenguaje con soporte para lógica temporal; o si se desean especificar requisitos para un sistema concurrente o distribuido, las opciones a considerar serían fundamentalmente CCS, CSP, SDL, ESTELLE o LOTOS.

Otro parámetro a tomar en cuenta es la naturaleza gráfica de los lenguajes. Por ejemplo, si se desea contar con representaciones visuales de las especificaciones que no dejen de ser rigurosas matemáticamente hablando, se deberían considerar las Redes Petri o STATECHARTS.

La madurez de las herramientas asociadas a cada lenguaje es otro factor que no se debe ignorar. Por ejemplo, PROMELA en conjunto con su herramienta SPIN son conocidos por su versatilidad en la verificación de sistemas. Por su parte, HOL tiene asociados diversos productos probadores de teoremas, y SDL también proporciona diversas herramientas destacadas.

También es importante conocer si el método elegido ha sido aplicado previamente en proyectos industriales reales. Por ejemplo, Z ha sido utilizado exitosamente en el proyecto CICS (*Customer Information Control System*) de IBM y por el Laboratorio de Computación de la Universidad de Oxford y en el proyecto CIDS para especificar formalmente el software del Airbus A330/340, por mencionar sólo algunos ejemplos.

Por su parte, VDM ha sido utilizado para verificar el sistema CDIS de control de tráfico aéreo de Londres y para especificar los requisitos de seguridad de ACS, un sistema de

almacenamiento de explosivos del ejército británico. SDL también ha sido utilizado para especificar requerimientos de un sistema de video llamado INSYDE, y en el proyecto NEWCORE (un sistema de telecomunicaciones) fue útil durante el proceso de verificación. Por su parte, el uso de lenguaje B fue fundamental para el desarrollo completo del software que controla la línea catorce del metro de París.

En el proceso de elección de un método formal, siempre resulta útil la opinión de otros desarrolladores que han utilizado previamente los lenguajes, con la finalidad de conocer sus experiencias y tomarlas en cuenta. Por ejemplo, en (KNIGHT, John et al., 1997) se describe la aplicación de tres métodos formales (Z, PVS y STATECHARTS) a un proyecto real que consistía en un sistema reactor de la Universidad de Virginia. Los expertos de dominio participantes, que eran ingenieros nucleares, opinaron que Z fue muy efectivo para la comunicación, pero su notación matemática no les resultaba familiar; acerca de PVS opinaron que tenía mucho parecido con el código fuente y que introducía demasiadas variables que los confundían; respecto a STATECHARTS comentaron que fue sencillo de aprender, muy útil para comunicar, pero difícil para realizar búsquedas y navegar entre los estados. Sin embargo, resaltaron que un diagrama de este tipo sería mejor que la actual documentación con la que contaban.

Por su parte, los expertos en cómputo que participaron en el proyecto comentaron que Z les resultó muy sencillo para entenderlo, navegar y realizar búsquedas; también les pareció un lenguaje razonablemente fácil de entender. Por el contrario, encontraron que PVS era difícil para entenderlo, para navegar y para realizar búsquedas; para algunos fue difícil de aprender, mientras que para la mayoría fue tan fácil, o más, como aprender un lenguaje de programación. Respecto a STATECHARTS, lo encontraron sencillo de entender, muy fácil de aprender, pero difícil para la navegación y las búsquedas.

Por último, los detalles particulares del proyecto, la experiencia de los desarrolladores, así como los recursos disponibles en ese momento también son factores a considerar en la elección de un método formal. Puede ocurrir que un sistema sea demasiado extenso y que abarque demasiados dominios, tales que resulte inviable utilizar un solo lenguaje formal. En este caso, convendría dividir el sistema y realizar la especificación separada de cada módulo utilizando el lenguaje más adecuado para ese propósito en particular.

Capítulo 3. Reutilización

3.1. ¿Qué es la reutilización de software?

La reutilización de software implica la utilización repetida en contextos diferentes de cualquier elemento necesario para el desarrollo de un sistema. Los elementos susceptibles de reutilización no se limitan al código fuente, también incluyen requerimientos, elementos de diseño, estructuras de implementación, especificaciones, documentación, casos y datos de prueba, transformaciones, y en general, cualquier producto de alguna actividad del desarrollo.

Se pueden diferenciar dos perspectivas diferentes de esta definición: el desarrollo con elementos reutilizables y el desarrollo de elementos reutilizables.

Cuando se desarrolla un sistema con elementos reutilizables, la actividad se centra en incorporar al nuevo proyecto elementos de software previamente elaborados en lugar de construirlos desde cero. Por el contrario, el desarrollo de elementos reutilizables se encarga de producir aquellos elementos con una previa planeación que facilite su posterior reutilización.

3.1.1. Breve evolución histórica de la reutilización de software

En 1968, McIlroy propuso una biblioteca de componentes reutilizables para cálculos numéricos, conversión de entradas y salidas, procesamiento de texto y almacenamiento dinámico, introduciendo así por primera vez el concepto de reutilización formal de software. La propuesta de McIlroy contemplaba un escenario donde era posible construir sistemas complejos a partir de componentes pequeños que estaban disponibles desde una librería. Esta idea sirvió de base para el concepto de "Fábricas de software" que en 1974 fue inscrito como marca registrada por Systems Development Corp. El concepto no hacía referencia explícita a la reutilización, pero establecía un conjunto de procedimientos repetibles para definir un proceso de software, donde se reutilizaba todo lo posible, incluyendo experiencia previamente adquirida.

Posteriormente, a mediados de los setenta, Robert Lanergan inició el primer proyecto de reutilización formal en una organización: Raytheon Missile Division. Para finales de los setenta, el interés de los investigadores hacia la reutilización se había incrementado notablemente dando lugar al primer taller internacional de reutilización (*First International Workshop on Reusability in Programming*).

La década de los ochenta estuvo marcada por un claro interés en la reutilización a nivel mundial en diferentes ámbitos. Por ejemplo, surgieron programas de reutilización a gran escala promovidos por los gobiernos de Estados Unidos y Europa, así como de grandes organizaciones en Japón (Hitachi, Toshiba, NEC, Fujitsu y NTT) y en Estados Unidos (GTE, AT&T, IBM y HP). También en esa época la definición de reutilización conoció nuevos límites gracias a Basili, que en 1988 la modificó para incluir cualquier elemento relacionado con un proyecto de software, incluso conocimiento. Para finales de esa misma década se observaron trabajos importantes que denotaron avances en las tecnologías relacionadas con la reutilización como: manejo de librerías, técnicas de clasificación, creación y distribución de componentes, así como programas de reutilización corporativos.

Poco después, ya en los noventa, Prieto-Díaz (PRIETO-DIAZ, Rubén, 1993) realizó entre otros trabajos, taxonomías importantes acerca de las diferentes formas de reutilización que aún hoy en día son aceptadas. A partir de esta década, también se resaltó la importancia de los factores económicos, administrativos, culturales y legales relacionados con la reutilización.

Los trabajos a partir del año 2000 reconocen la reutilización como un proceso institucional, y bien definido. Hasta la fecha se siguen realizando propuestas para mejorar y aplicar las tecnologías asociadas a la reutilización, y se conducen estudios para conocer mejor sus tendencias de implantación y adopción en las organizaciones de desarrollo de software.

3.1.2. Motivación para la reutilización

Desde sus inicios, se ha visto la reutilización como una manera de enfrentar la “crisis del software”, identificada por primera vez a finales de los sesenta como la causa de proyectos de mala calidad: defectuosos, retrasados y caros. Algunos factores que contribuyen a este problema son: la naturaleza cambiante de las necesidades de los usuarios, la magnitud de los proyectos, así como la complejidad del área del sistema a desarrollar y de la gestión del proceso mismo.

Al reutilizar diversos elementos de software cuya calidad y funcionalidad previamente fue comprobada, se puede disminuir el tiempo, el costo y el esfuerzo de desarrollo; con lo que es posible centrar la atención en satisfacer los requerimientos de los usuarios con la certeza de que los componentes utilizados son confiables y funcionales.

Utilizando metodologías adecuadas, la reutilización puede ayudar a entregar productos de buena calidad en menos tiempo ya sea en proyectos pequeños o a gran escala.

3.1.3. La paradoja de la administración de la reutilización de software

Es bien sabido que al reutilizar elementos de software se obtienen beneficios económicos; aunque no inmediatos, sí a medio y largo plazo. Sin embargo, al implementar un programa de reutilización, se debe prever con una alta inversión inicial adicional, relacionada con la obtención del software reutilizable, el uso del software obtenido, y la implantación del proceso en sí.

La paradoja de la administración de la reutilización de software (ver Figura 3.1), es el resultado de un proyecto realizado en Nippon Telegraph and Telephone Corporation (NTT) durante cuatro años, que exhibe la dificultad de obtener ganancias de la reutilización de software sin una inversión considerable de tiempo y esfuerzo; y por otra parte, muestra que es muy difícil justificar considerables cantidades de tiempo y esfuerzo sin ganancias tangibles a corto plazo. Éstas y otras conclusiones obtenidas del estudio conducido en NTT (ISODA, Sadahiro, 1995) se resumen en el Cuadro 3.1.

Por otra parte, para incorporar un programa de reutilización, además de los factores financieros también existen factores organizacionales que impactan directamente la forma de trabajo. Por ejemplo, la estructura organizacional debe sufrir cambios, ya que el personal se debe re-acomodar a las funciones y equipos de trabajo; también los procedimientos tradicionales se deben re-ajustar para incluir tareas propias de la reutilización.

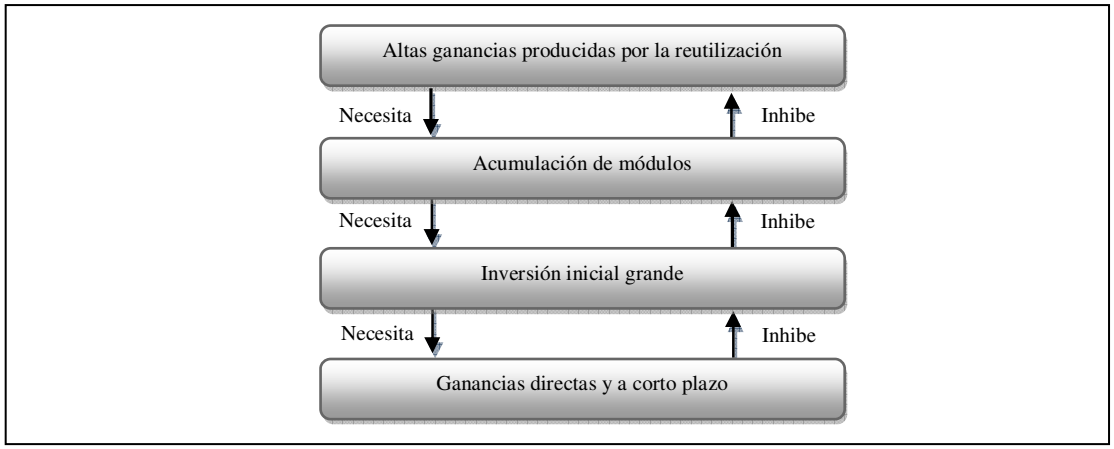


Figura 3.1. La paradoja de la administración de la reutilización de software. (MILI, Hafedh et al., 2002)

Para…	…un programa de reutilización debe…
Maximizar la reutilización…	…enfocarse en un dominio de aplicación restringido con muchas características comunes.
Amortizar los costos de inversión…	…involucrar desarrollo de software en gran volumen.
Justificar el tiempo requerido para introducir los procesos propios de la reutilización y poblar la biblioteca de componentes…	…planearse por largo tiempo.
Mantener los módulos en uso…	…aplicar estrictos criterios para incluir componentes en la biblioteca.
Mantener un control adecuado de costos, beneficios y tendencias asociados con la reutilización…	…tener definidas sus métricas apropiadamente.
Asegurar que a la iniciativa de reutilización se le está dando el tiempo y los recursos necesarios para madurar y producir resultados…	…contar con el apoyo de la dirección de alto nivel.

Cuadro 3.1 Conclusiones acerca de la administración de software.

3.1.4. Tipos de reutilización

En el trabajo de (PRIETO-DIAZ, Rubén, 1993) se clasifican las formas de analizar la reutilización tomando en cuenta diferentes perspectivas: por esencia, por ámbito, por modalidad, por técnica, por intención y por producto (ver Cuadro 3.2). Cada categoría se describe a continuación.

Por esencia	Por ámbito (dominio)	Por modalidad	Por técnica	Por intención	Por producto
- Ideas, conceptos - Elementos y Componentes - Procedimientos y habilidades	- Vertical - Horizontal	- Planeada y Sistemática - Ad-hoc u Oportunista	-Por composición -Por generación	- Caja negra - Caja blanca	- Código fuente - Diseño - Especificaciones - Objetos - Texto - Arquitecturas

Cuadro 3.2 Clasificación de la reutilización. Adaptado de (Prieto-Diaz, 1993).

Una perspectiva toma en cuenta la esencia de los elementos, y abarca las ideas y conceptos que representan soluciones generales que pueden ser aplicadas de nuevo; también incluye procedimientos y habilidades, así como elementos y componentes concebidos como parte de un todo.

Otra perspectiva de estudio considera el dominio de la aplicación del sistema; en este sentido, se puede hablar de reutilización horizontal y vertical. La vertical se produce entre sistemas que pertenecen al mismo dominio, en cuyo caso usualmente hay un alto grado de reutilización. La horizontal se produce entre sistemas de diferentes dominios, por lo que la reutilización se observa en menor grado.

El nivel de planeación con el que se practica la reutilización da lugar a una perspectiva de estudio más, se trata de la modalidad en el proceso: se puede identificar la reutilización planificada cuando ésta es sistemática y con procedimientos bien definidos, y la reutilización oportunista cuando ésta es sólo una práctica informal.

Considerando las técnicas utilizadas para llevar a cabo el proceso, existen dos tipos de reutilización: por composición y por generación (ambas abordadas en los siguientes apartados). La primera se refiere a incorporar en los nuevos sistemas, aquellos elementos existentes que se encuentran encapsulados en una sola entidad identificable. La segunda es un tipo particular de reutilización en la que los componentes no se encuentran auto-contenidos en un solo lugar, sino que son el resultado de un proceso que produce ciertas codificaciones pre-establecidas en función de diversas condiciones de entrada.

Una nueva perspectiva de estudio en las maneras de reutilizar surge si se considera la intención al integrar los componentes en el nuevo sistema. Bajo este enfoque se pueden diferenciar elementos de caja negra, de caja blanca o de caja transparente. Si un elemento actúa como caja negra, se conoce su interface y funcionalidad pero no su contenido, por lo que no es posible modificarlo. Si un elemento es una caja blanca, es posible realizarle modificaciones para alterar su funcionalidad. Por otra parte, las cajas transparentes permiten conocer su contenido, pero lo restringen para no aceptar modificaciones.

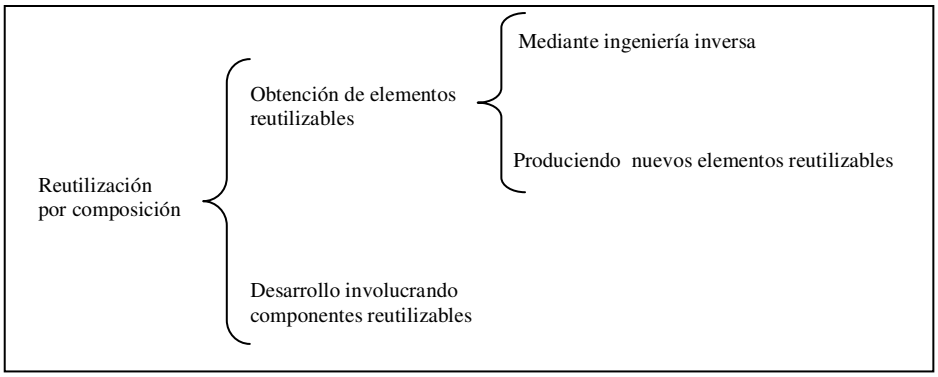
Finalmente, de acuerdo a los elementos o productos que pueden usarse de nuevo, se identifican: requisitos, especificaciones, arquitecturas, documentación, código, pruebas, etc. Estos productos pueden reutilizarse si se cuidan ciertas condiciones de compatibilidad entre ellos y los elementos existentes en los proyectos donde se incorporan. Por ejemplo, al momento de integrar un elemento de diseño en un proyecto, es deseable que ambos tengan semejanza en el grado de abstracción de sus respectivas representaciones.

3.2. Reutilización por composición

La reutilización por composición resumida en el Cuadro 3.3 consta de dos actividades: la obtención de elementos reutilizables, y el desarrollo involucrando componentes reutilizables. Ambas actividades requieren de una biblioteca donde se almacenan los componentes y desde donde posteriormente se accede a ellos.

La biblioteca es un concepto similar a un depósito o repositorio de elementos reutilizables. Se desea que los componentes reutilizables dentro de la biblioteca sean "*auto-contenidos, describiendo una funcionalidad clara y documentada, proporcionando una interfaz apropiada y un registro o estado de reutilización*" (DIAZ, Rebeca, 2002) y que además, tengan asociada la información acerca de su interfaz, cuerpo, documentación, clasificación, pruebas, implementación, adaptación y administración, con la finalidad de que cada componente posea una descripción lo más completa posible acerca de sí mismo y del entorno en el que se puede reutilizar.

La obtención de elementos reutilizables puede hacerse de dos maneras: extrayendo elementos de sistemas existentes, es decir, aplicando ingeniería inversa o bien, produciendo nuevos elementos a partir de cero.



Cuadro 3.3 Reutilización por composición.

Cuando se tienen sistemas existentes, también llamados heredados, típicamente se aplica ingeniería inversa, la cual se encarga de elaborar elementos de las primeras fases del ciclo de vida a partir de elementos propios de las últimas fases. De esta manera, por ejemplo, es común tomar código y generar las especificaciones correspondientes. Al obtener componentes de un sistema heredado, se consideran dos fases: la identificación y la calificación de elementos.

Durante la fase de identificación se obtienen los elementos que pueden ser reutilizables y se les realiza una evaluación para tal efecto. Solamente si superan esta prueba se pasa a la fase de calificación, donde se estudia el componente para comprenderlo, documentarlo y representarlo formalmente. La fase de calificación se subdivide en las siguientes actividades para cada elemento analizado: primero, se obtiene la especificación formal de su funcionalidad; posteriormente se genera un conjunto de pruebas para comprobar su correctitud; luego se incluye su documentación correspondiente; y finalmente se clasifica y se almacena en una biblioteca de componentes.

Independientemente de las distintas metodologías disponibles que se pueden adoptar al generar nuevos elementos reutilizables, se deben analizar siempre las opciones para generar cada componente y seleccionar la más conveniente ya sea por sencillez, comprensión, o validación; se deben identificar las funciones que se le pueden añadir, estudiando siempre la factibilidad de hacerlo; y finalmente, se debe establecer la interfaz más apropiada para ese componente.

Una vez generado el componente reutilizable, se debe realizar una certificación del mismo. Este proceso sirve para verificar el cumplimiento de los requisitos mínimos de calidad. Un componente se puede certificar de diferentes maneras. Por ejemplo, en la certificación por niveles, cada nivel está relacionado con el número de requisitos que satisfacen los

componentes que pertenecen a él. Otra manera es por certificación local, donde se toma en cuenta la verificación formal de cada elemento sin considerar contexto alguno. Una opción más es la certificación orientada al dominio, en donde los requisitos de calidad de los componentes dependen de los requisitos de calidad de las aplicaciones, especificaciones, etc.

Una vez que el componente ha sido certificado, se almacena en una biblioteca, la cual se mantiene con el objetivo de organizar, almacenar y extraer todos los componentes contenidos en ella. Una biblioteca se considera local si contiene componentes de propósito general para usarse en distintas aplicaciones; se considera específica de un dominio si almacena componentes de una sola área de aplicación. En cualquier caso la biblioteca puede contener los componentes directamente, o bien referencias o ligas a la ubicación real de los componentes.

Dentro de la biblioteca, los componentes deben clasificarse utilizando una o varias técnicas. Las técnicas de clasificación más comunes son basadas en inteligencia artificial, o en técnicas bibliotecarias. Las técnicas basadas en inteligencia artificial proporcionan sistemas de clasificación muy flexibles. Las técnicas bibliotecarias realizan búsquedas por texto libre que se extrae de la representación de sus elementos asociados, o por palabras clave que describen al componente, o por clasificación de categorías y sub-categorías, o por facetas, donde se asocian distintas facetas o características independientes a cada componente, con sus respectivos valores.

Otra de las tareas que se realiza en la biblioteca de componentes es la recuperación de elementos previamente almacenados, para lo cual: primero se deben identificar las características buscadas; acto seguido se debe armar un patrón de búsqueda, el cual, al ejecutarse debe regresar todos los elementos que cumplan con el criterio solicitado. En base a las características funcionales y no funcionales de cada componente incluido en el resultado de la búsqueda, se deben seleccionar los elementos más apropiados, y evaluar sus posibilidades en relación con los requisitos del usuario que se busca satisfacer con el componente. Finalmente se selecciona la mejor opción y se adapta e integra al nuevo sistema.

3.3. Reutilización por generación

Este tipo de reutilización implica la creación de un programa generador, el cual se puede definir como una herramienta o un conjunto de herramientas integradas que recibe como entrada un conjunto de especificaciones y genera código de aplicación en un lenguaje de implementación (MILI, Hafedh et al., 2002). La reutilización por generación ha tenido tres grandes ramificaciones: compiladores de sistemas, generadores de aplicaciones, y procesos transformacionales.

Los compiladores de sistemas utilizan un lenguaje de muy alto nivel (VHLL –*Very High-Level Language*-) para expresar especificaciones y posteriormente generar el ejecutable correspondiente. Algunos ejemplos son: SETL (SCHWARTZ, J. T. et al., 1986), PAISLey (ZAVE, Pamela, 1991) y MODEL (PRYWES, N. S. et al., 1979).

Los generadores de aplicaciones son muy parecidos en funcionalidad a los compiladores, pero se utilizan para un dominio específico, además, pueden extender el lenguaje y realizar transformaciones dentro de las estructuras de los programas. Contrariamente a un compilador, en un generador las especificaciones de entrada son parciales, y la herramienta se encarga de completarlas con valores predeterminados; además, el formato de las entradas puede ser declarativo, gráfico o procedimental.

Cuando se desea elaborar un generador de aplicaciones, primero se debe identificar el dominio de las aplicaciones para las cuales va a ser útil. Posteriormente, se enumeran las características que debe poseer, y se debe definir la manera en la cual se podrán realizar las especificaciones de los sistemas. Finalmente, se debe desarrollar e implementar el generador. El lenguaje de especificación y la técnica de generación utilizados dependen fuertemente del dominio de aplicación; por eso es difícil establecer una metodología universal para todos los casos.

Se han identificado tres principales dificultades en la creación de generadores: El primer problema es reconocer los casos en que realmente es necesario construir uno; la segunda dificultad radica en definir sus requerimientos de entradas y salidas y los algoritmos de transformación; y la última y tal vez más importante, es la necesidad de validación de sus salidas, que dicho de otra manera significa asegurarse que el código generado haga lo que tiene que hacer.

Por otra parte, los sistemas transformacionales permiten que una especificación inicial sea modificada sucesivamente hasta llegar a un sistema final. Una transformación consiste en una sustitución de un elemento por otro, y puede tratarse de un refinamiento o de una optimización. La diferencia entra ambas es que un refinamiento agrega detalles a una especificación mientras que una optimización se concentra en mejorar la eficiencia de la tarea a realizar. En realidad, las transformaciones no se limitan a estas categorías, por el contrario, pueden seguir especializándose. Por ejemplo, los refinamientos pueden aplicarse a algoritmos o a tipos de datos; mientras que las optimizaciones pueden ser de evaluación parcial si especializan un fragmento de código generalizado, o incrementales si unifican varias operaciones parciales correspondientes a una operación compleja, o de re-estructuración de código si realizan una mejora al código.

Es difícil que un proyecto que involucra la reutilización por generación pertenezca exclusivamente a una sola ramificación de las tres mencionadas (compiladores, generadores de aplicaciones o sistemas transformacionales); típicamente el resultado involucra una mezcla de ideas que toma algunas características de cada rama. Independientemente del enfoque utilizado, un software generado con reutilización por generación reduce el ciclo de vida tradicional. Por ejemplo, si se utiliza un generador de aplicaciones, el ciclo de vida se reduce a la fase de especificación, pues el ejecutable se obtiene automáticamente a partir de

ella. Mientras tanto, en un entorno transformacional, se utilizan dos fases: la de especificación del sistema y la de transformaciones sucesivas que da como resultado el sistema final.

3.4. Tendencias en la reutilización

Cada uno de los diferentes enfoques de reutilización de software se puede concebir como un punto en un espacio de tres dimensiones: organizacional, gerencial y técnica; cada una de estas dimensiones representa un tipo de aspecto importante que los enfoques deben considerar. Siguiendo esta perspectiva, se obtendría en ese espacio tridimensional, un número infinito de puntos, es decir, de enfoques de reutilización, y resultaría muy difícil explorar cada uno en profundidad en la práctica. Afortunadamente, han surgido ciertas tendencias que reducen el número de posibilidades y corresponden a conjuntos particulares de configuraciones en las tres dimensiones mencionadas.

Las tres tendencias más importantes son: la Ingeniería de software basada en componentes (CBSE –*Component Based Software Engineering*-), la Ingeniería de líneas de productos (PLE -*Product Line Engineering*-) y el Desarrollo basado en componentes listos para utilizarse (COTS -*Component Off The Shelf Based Development*-). A continuación se aborda cada uno de ellos.

3.4.1. *Component Based Software Engineering* (CBSE)

La ingeniería del software basada en componentes (CBSE) es una manera de reutilización de software que incorpora elementos especialmente encapsulados denominados componentes en el desarrollo de nuevos sistemas mediante el enfoque de reutilización de caja negra.

Los componentes proporcionan servicios bien definidos a las aplicaciones que los utilizan. Son elementos auto-contenidos, claramente identificables, con funciones específicas, independientes de la implementación y que requieren poca o ninguna adecuación. Si un componente requiere cierta adecuación, se deben ajustar los valores de algunas de sus propiedades antes de su uso en las aplicaciones, pero sin modificar el contenido del componente en sí.

En cuanto a la implementación, la CBSE es un proceso centrado en las interfaces y la composición, que requiere de técnicas importantes de abstracción. Conocer las interfaces de los componentes es importante para poder incorporarlos sin problemas a otras aplicaciones mediante técnicas de composición, mientras que la abstracción correcta es indispensable para una representación y encapsulación adecuada de los elementos dentro de un componente.

3.4.1.1. Componentes

Un componente es una unidad coherente estratégicamente encapsulada con interfaces especificadas y dependencias explícitas, que puede ser desarrollado y distribuido de forma independiente, pensando en que será parte de un proceso posterior de composición con otras aplicaciones. Los componentes deben cumplir con una interface estándar y bien definida que debe ser conocida por las aplicaciones que van a utilizarlos. Las interfaces indican los servicios que proporciona el componente y definen su manera de interactuar con otros componentes y aplicaciones. Por otra parte, un componente no está restringido a ser implementado mediante la programación orientada a objetos, aunque ésa sea una posibilidad. Un componente tampoco es necesariamente un objeto, pues típicamente es más grande que una clase.

Comúnmente, un componente se describe en términos de sus interfaces. Para tal propósito se suele utilizar un lenguaje existente, por ejemplo Distributed Connection Language (ASTLEY, Mark and Agha, Gul, 1998) y Microsoft Interface Description Language (TROELSEN, Andrew, 2000). La anatomía de un componente consta de elementos internos, interfaces de aplicación e interfaces con la plataforma. Estos elementos y sus relaciones se muestran en la Figura 3.2 y se explican a continuación:

- Elementos internos.- Se refiere a la estructura privada del componente que se encarga de implementar la funcionalidad ofrecida en sus interfaces. Estos elementos permanecen ocultos del exterior del componente.

- Interfaces de aplicación.- Las interfaces de aplicación o de canal horizontal interactúan con otros componentes o aplicaciones y describen la estructura de los mensajes intercambiados entre ellos. Existen dos tipos de interfaces de aplicación: aquellas que representan la funcionalidad que el componente provee y aquellas que indican la funcionalidad que el componente requiere de los componentes externos.

- Interfaces con la plataforma.- Las interfaces con la plataforma, llamadas también de canal vertical, definen la interacción del componente con la plataforma sobre la que se ejecuta; es decir, describen los mensajes con el hardware, procesador, memoria, sistema operativo, entorno de ejecución, acceso a los periféricos, sistemas de comunicación, etc.

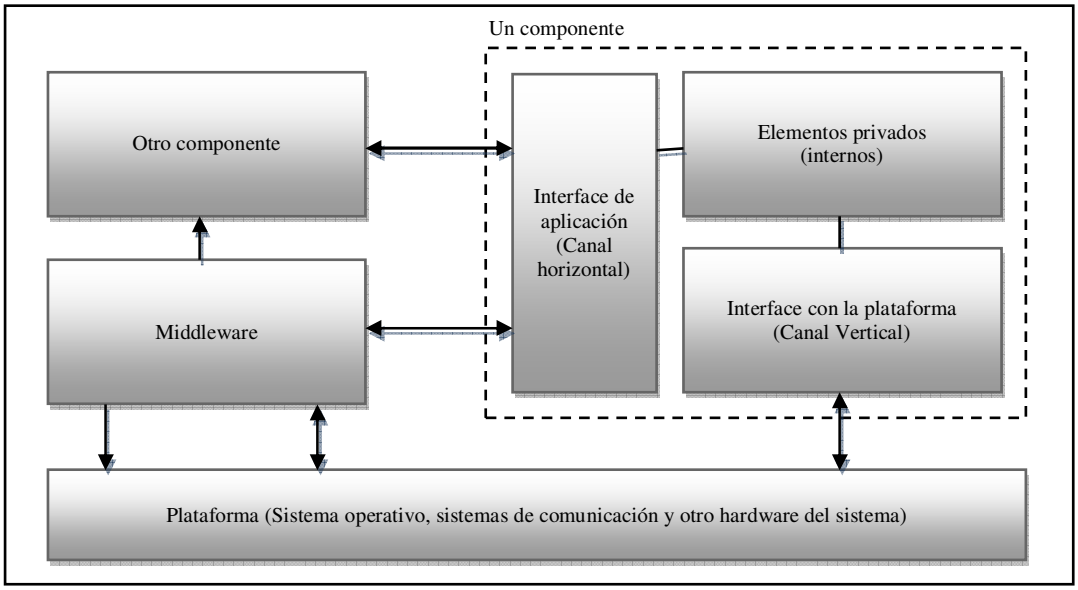


Figura 3.2. Anatomía de un componente mostrando sus interacciones. (MILI, Hafedh et al., 2002).

Un buen componente debe tener las características de todo elemento reutilizable, es decir, debe ser: bien documentado, cohesivo, débilmente acoplado, útil y debe estar certificado. Además, por ser de naturaleza ejecutable o binaria y su modo de reutilización de caja negra, también debe estar apegado a un modelo de componentes, con lo cual asegura que pueda ser compuesto con cualquier otro elemento que pertenezca al mismo modelo. Además, debe ser seguro, con lo cual controla los accesos hacia sus elementos internos. Esta última característica adquiere especial importancia en la CBSE pues se debe tener en mente que al ser integrados en los proyectos, los componentes son invocados remotamente desde otras aplicaciones, por lo que sus mecanismos de seguridad deben ser suficientemente robustos.

3.4.1.2. Modelos de componentes

Un modelo de componentes es un marco para definir componentes dentro del contexto de un estilo de arquitectura de software en particular. El modelo es el encargado de definir tipos de componentes y los requerimientos de esos tipos para ofrecer sus servicios o consumir los de otros componentes que hayan sido desarrollados utilizando este modelo. Un modelo de componentes especifica las interfaces y la semántica necesaria para implementarlas. Algunos ejemplos de modelos son: JavaBeans (RUSTY HAROLD, Elliotte, 1998), CORBA y COM (PRITCHARD, Jason, 1999).

Un modelo de componentes debe definir las características a las que se deben apegar los componentes desarrollados dentro de ese modelo. Algunos aspectos de los que se debe ocupar son:

- Representación de datos.- Se refiere a los tipos de datos permitidos en el intercambio de datos de dos componentes.
- Transferencia de datos.- Se refiere a la dirección del flujo de datos y la secuencia de las operaciones de transferencia de datos entre componentes.
- Protocolo de transferencia.- Describe la manera en que los datos se transfieren entre los componentes y especifica cómo deben interactuar los componentes para realizar una transferencia.
- Persistencia de estado.- Es la manera de acceder los valores de las características de un componente en un momento determinado. Puede ser que un componente tenga o no información disponible de su estado.
- Ámbito del estado.- Define el grado de visibilidad desde el exterior del estado interno del componente.
- Fallos.- Se refiere a la manera para reportar los errores que puedan aparecer en un componente.
- Establecimiento de la conexión.- Se refiere a la manera en la que una conexión entre componentes se inicia y se termina.

3.4.1.3. Ciclo de vida CBSE

De manera general se pueden identificar las siguientes actividades en el desarrollo basado en componentes (ver Figura 3.3). La manera de realizar cada una depende de la naturaleza y el tipo de componente.

- Encontrar.- Durante esta actividad, los ingenieros de dominio realizan minería sobre las familias de productos similares para documentar los principales componentes base.
- Seleccionar.- Se seleccionan componentes específicos para utilizarlos en el desarrollo de software basado en componentes.
- Adaptar.- Se adecúan los componentes previamente seleccionados con el objetivo de satisfacer los requerimientos de usuario dentro del contexto en el que se va a utilizar el componente.
- Crear.- Esta actividad se realiza cuando la adaptación no fue suficiente para cumplir con los requerimientos necesarios. En este caso, se deben desarrollar desde cero nuevos componentes para esa aplicación.
- Componer.- Es el proceso de ensamblaje e integración de componentes dentro del nuevo sistema.
- Reemplazar.- Esta actividad se lleva a cabo cuando existen versiones antiguas de un componente que deben ser modificadas por una más nueva.

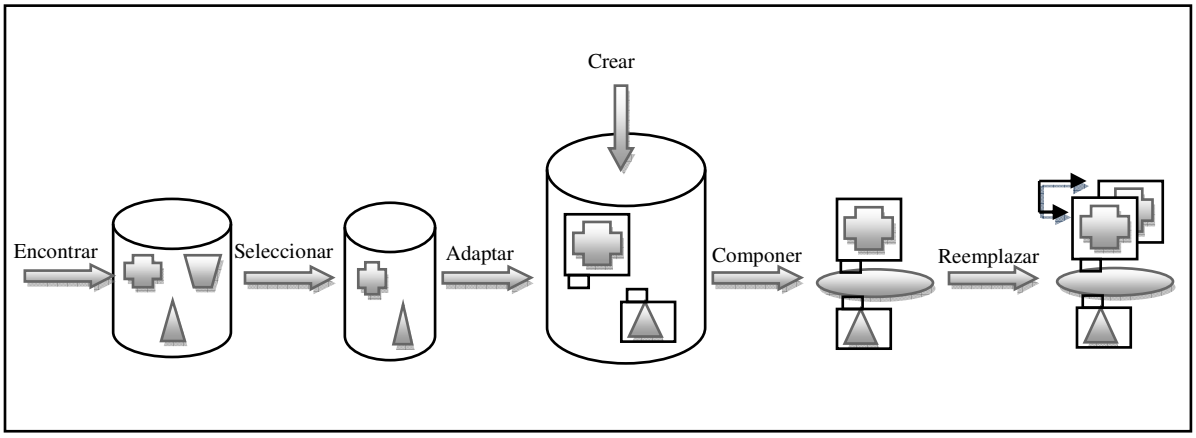


Figura 3.3. Actividades en el desarrollo basado en componentes. (MILI, Hafedh et al., 2002).

3.4.2. *Product Line Engineering* (PLE)

La ingeniería de línea de productos es una manera de reutilizar software que promueve la incorporación de los elementos comunes de un dominio en distintos proyectos. Mediante este enfoque también se seleccionan, refinan y establecen prácticas para identificar y explotar las características similares que existen entre productos de un dominio particular (MILI, Hafedh et al., 2002). Si es correctamente implementada, la PLE resulta en productividad, calidad y tiempo corto de entrega entre productos. Además, al realizar una sola inversión en una infraestructura que permite sacar provecho de los productos en varios proyectos, se acelera la recuperación y los beneficios económicos, al mismo tiempo que se reduce el costo de los proyectos individuales.

El enfoque de PLE considera que cada nuevo proyecto de software no es un esfuerzo aislado independiente de los productos y experiencias de otros proyectos, sino una iniciativa que aprovecha las características comunes de un grupo de sistemas construidos anteriormente dentro del mismo contexto. Los elementos reutilizables obtenidos ya sea por reingeniería o por nuevo desarrollo también deben mantener siempre una visión que considere su uso en otras nuevas aplicaciones. La filosofía de PLE también considera que la creación de elementos totalmente genéricos que puedan reutilizarse en cualquier situación rara vez puede ser implementada por completo. Por eso, contrariamente a otros enfoques de reutilización, el dominio se restringe para incluir solo un grupo de aplicaciones.

En una línea de productos se desarrollan muchos productos de software, y aunque cada uno tiene su propio ciclo de actividades, entrega y versión, todos comparten una arquitectura de dominio común. La evolución de cada producto debe ser cuidada considerándose como parte de la evolución de la línea completa y no como un hecho aislado; esto es indispensable para que la versión de los productos individuales no afecte a la arquitectura

del dominio común. Una línea de productos requiere de tres elementos imprescindibles para poder implementarse: análisis de dominio, arquitectura de software y un proceso de desarrollo.

- Análisis de dominio.- Su principal objetivo es obtener un modelo de dominio que incluya las características comunes y variaciones entre los miembros de la familia de la línea de productos.
- Arquitectura de software.- Es una infraestructura estándar que representa un esqueleto común, el cual es adoptado como arquitectura propia por cada producto de la línea.
- Proceso de desarrollo.- El proceso de desarrollo divide las actividades en dos grandes grupos: para la ingeniería de dominio y para la ingeniería de aplicación. Las actividades del proceso se abordan a continuación.

3.4.2.1. Ciclo de vida PLE

El ciclo de vida PLE contiene actividades de la ingeniería de dominio y de la ingeniería de aplicación (ver Figura 3.4). La ingeniería de dominio gestiona las actividades y productos que engloban la arquitectura común de los productos de la línea, mientras que la ingeniería de aplicación se encarga del desarrollo individual de cada producto a partir de los elementos del dominio común generados en la ingeniería de dominio.

Las actividades de la ingeniería de dominio son: el modelado, que comprende el análisis y definición del dominio, el diseño y la definición de una arquitectura común para el dominio y el desarrollo de componentes también llamado implementación de dominio. La ingeniería de aplicación incluye: análisis de los requerimientos de la aplicación, instanciación de la arquitectura base, instanciación y adecuación de los componentes del dominio y el subsecuente desarrollo utilizando los componentes del dominio.

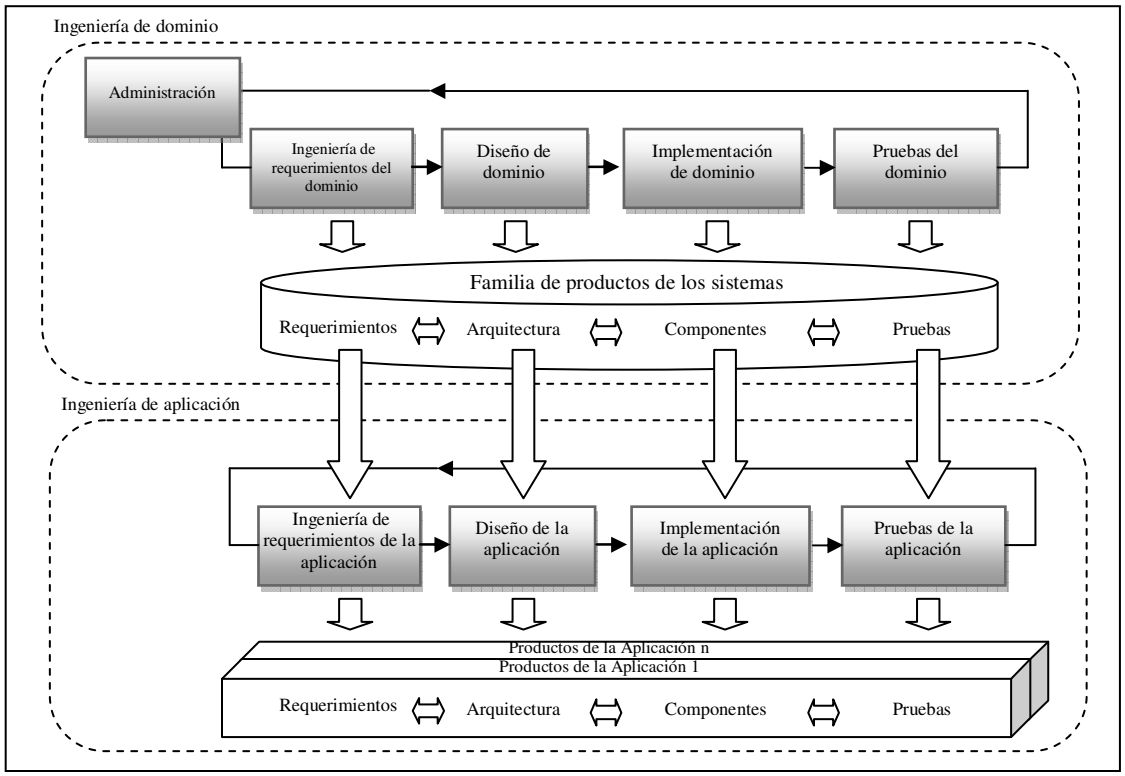


Figura 3.4. Ingeniería de dominio e Ingeniería de aplicación. (LINDEN, Frank van der et al., 2007).

El desarrollo de la arquitectura es un paso clave para la implementación exitosa de una línea de productos. Para tal efecto, los ingenieros de dominio analizan los modelos de dominio y establecen una arquitectura común, la cual posteriormente se instancia en la ingeniería de cada aplicación para convertirse en la arquitectura individual de cada producto desarrollado, una vez que ya se han tomado en cuenta sus especificaciones propias. Las decisiones que se tomen referentes a la arquitectura en las etapas tempranas, impactarán de manera directa la calidad de la familia completa de productos. Si la arquitectura no es la apropiada, los requerimientos del sistema no serán satisfechos completamente o la variación entre los productos no estará bien definida. Se requiere de una arquitectura robusta, flexible y altamente configurable para producir software de alta calidad, aumentar la productividad y garantizar un buen desarrollo y mantenimiento de los sistemas desarrollados.

Como una analogía, el proceso de instanciación al que se hace referencia en PLE es similar en concepto al proceso de creación de objetos en la Programación Orientada a Objetos, en donde a partir de una clase se crean múltiples objetos con estados independientes pero con estructura e interface similares. En PLE, a partir de una definición base, también se crean o instancian múltiples elementos independientes y configurables que comparten la estructura común especificada en la arquitectura de dominio. Básicamente, hablar de instanciación, significa:

- Tomar los componentes y conectores comunes de la arquitectura del dominio.
- Identificar los elementos de arquitectura que son específicos del producto que está desarrollándose.
- Validar que la arquitectura de aplicación del producto específico corresponda con la arquitectura de dominio.

Cuando se realiza la instanciación de un producto, se debe poner especial atención a dos situaciones: la correspondencia entre las arquitecturas base e instanciada, y la sincronización de las versiones de ambas arquitecturas. En la Figura 3.5 se ubican y se relacionan ambas; a continuación se explica en qué consiste cada una:

- Medir la correspondencia entre la arquitectura instanciada y la arquitectura base.- Se refiere a determinar en la instancia el grado de satisfacción de las restricciones impuestas por la arquitectura de dominio.
- Sincronizar las nuevas versiones de la arquitectura base con las versiones de los productos.- Significa monitorear los cambios realizados en la arquitectura base para no afectar a los productos individuales. Una arquitectura de la línea de productos puede evolucionar para incorporar cambios tecnológicos, solucionar problemas o agregar nuevas funcionalidades. Esta evolución puede causar problemas a la configuración individual de los productos, de ahí la importancia de mantener una adecuada sincronización.

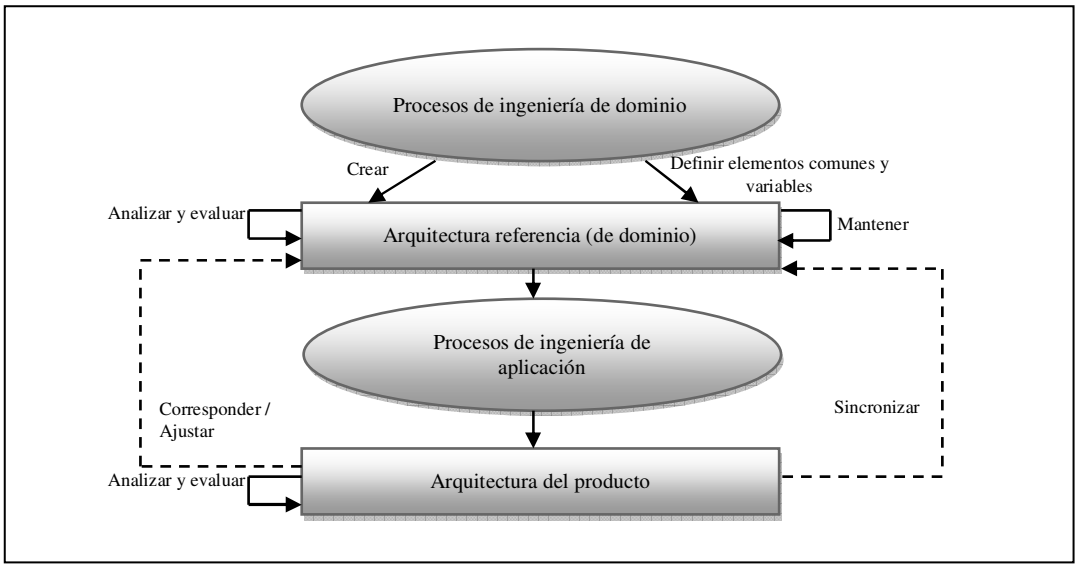


Figura 3.5. Arquitecturas de dominio y de producto. (MILI, Hafedh et al., 2002).

3.4.3. *Component Off the Shelf Based Development* (COTS)

El desarrollo basado en componentes comerciales (COTS) es un proceso para construir aplicaciones mediante la integración de componentes disponibles comercialmente que fueron desarrollados por terceros. Un producto desarrollado mediante COTS es un ejecutable con múltiples copias idénticas que se vende o se permite usarlo bajo algún tipo de licenciamiento al público en general. Quienes lo adquieren no tienen acceso al código fuente; quien lo vende usualmente es quien lo desarrolló y quien se encarga de su actualización y mantenimiento.

El desarrollo COTS posee varias ventajas. Por ejemplo, ofrece ganancias en costo, pues se desarrolla una sola vez y se utiliza muchas veces; en calidad, pues usualmente ha sido exhaustivamente probado y depurado para ser usado por múltiples usuarios; en funcionalidad, porque sus desarrolladores suelen ser expertos en el dominio del componente; en rapidez de desarrollo, porque una vez adquirido se puede utilizar inmediatamente; y en mantenimiento, pues el vendedor es el encargado de solucionar los problemas que surjan).

Por otra parte, el desarrollo COTS implica ciertas desventajas. Por ejemplo, un componente COTS no puede ser usado en sistemas con estrictos requerimientos de confiabilidad ya que pueden existir diferencias entre la funcionalidad que proporciona el componente y la que se espera de él al momento de reutilizarlo. Además, es posible que el componente no se comporte como se espera bajo ciertas condiciones, pues al momento de reutilizarlo se desconoce su estructura interna y al momento de desarrollarlo se desconocen todos los posibles escenarios en los que se puede incorporar. También, se debe considerar que una aplicación desarrollada usando componentes COTS no posee gran integridad o simplicidad, pues su proceso de desarrollo se enfoca más a obtener un buen resultado de la composición de elementos previamente desarrollados que en lograr un buen diseño.

3.4.3.1. Ciclo de vida COTS

El ciclo de vida de una aplicación desarrollada mediante el enfoque COTS típicamente tiene cuatro fases: selección, integración, verificación-validación, y mantenimiento. Se considera que los requerimientos particulares de la aplicación ya han sido analizados, capturados y especificados antes de iniciar con la primera fase de selección.

- Selección.- Significa escoger los componentes adecuados de un conjunto de candidatos. Para realizar la elección, se deben considerar atributos de calidad como eficiencia, confiabilidad, facilidad de integración y actualización, también se debe evaluar la aptitud del componente para esa aplicación y su correspondencia con los requerimientos, y finalmente, también se debe asegurar que el componente no tenga interacción indeseada con otros elementos del sistema en desarrollo.

- Integración.- Se refiere a incorporar el componente con la aplicación que lo utiliza. En esta fase es importante verificar que los elementos a integrar tengan interfaces compatibles, que tengan una afinidad en sus funciones y que el componente no tenga funcionalidades adicionales que puedan degradar o comprometer el desempeño de la aplicación.
- Verificación y validación.- Se refiere a evaluar el componente COTS dentro de la nueva aplicación. En esta fase se debe asegurar que el componente cumple con los requerimientos específicos que demanda la aplicación y que existe coherencia en el seguimiento de uso del componente entre las distintas fases del desarrollo de la aplicación. Las pruebas que se realizan en esta fase son de caja negra, dada la naturaleza de la filosofía COTS.
- Mantenimiento.- Se refiere a la liberación de nuevas versiones del componente por parte del proveedor. Ésta es la fase más crítica del ciclo de vida, pues se depende en gran medida de las acciones del vendedor del componente, quien es el responsable de proporcionar el mantenimiento a su producto. Bajo este esquema, los ingenieros de aplicación no tienen el control de los calendarios, costos y nuevas funcionalidades asociadas con la nueva versión del componente. Así, es común que un vendedor cobre adicionalmente por un componente que ofrece funciones adicionales, algunas veces innecesarias, en una versión que tarda en estar disponible. Cada nueva versión que se incorpore a las aplicaciones debe ser exhaustivamente probada para asegurarse de que es completamente compatible con el sistema y que no le agrega nuevos problemas. Una situación que también puede ocurrir es que el vendedor se retire del negocio y no ofrezca nuevas actualizaciones a su producto. Este caso se puede aminorar firmando un acuerdo legal que estipule que el comprador puede tener acceso al código fuente si el vendedor del componente deja de ofrecerlo comercialmente.

3.5. Conclusiones

La reutilización es un medio para lograr que la experiencia lograda en cada proyecto sea aprovechada en otros proyectos, reduciendo así el tiempo y el costo de desarrollo. Como conclusión, se compararán las tres tendencias de reutilización más importantes presentadas en este capítulo: CBS, PLE y COTS. Aunque las tres son formas especializadas de reutilización de software y comparten similitudes, entre sí también tienen marcadas diferencias.

Tal vez los enfoques más parecidos a simple vista sean COTS y CBSD pues ambos están basados en la composición. Sin embargo, los componentes COTS son siempre desarrollados por vendedores externos, y en CBSD los componentes son desarrollados por un ingeniero de la misma organización. Como consecuencia, CBSD es un enfoque que soporta reutilización de caja negra y de caja blanca, pues el código fuente está disponible localmente, mientras que el desarrollo COTS es típico de caja negra. Respecto a los derechos reservados, en COTS el propietario es el vendedor y en CBSD es el desarrollador

mismo de la aplicación. En cuanto al mantenimiento, el desarrollador de CBSD es el encargado de proporcionarlo, mientras que en COTS es responsabilidad del vendedor del componente. Por otra parte, la reutilización por generación puede ser implementada mediante CBSE y PLE, pero no mediante COTS, dado que los componentes comerciales usualmente son muy generales y no están desarrollados con la especialización requerida por un generador.

Las actividades cotidianas cambian en cada enfoque. Por ejemplo, en CBSE es común enfrentarse a un escenario donde sea necesario adaptar componentes, mientras que en PLE lo más usual es crear instancias de la arquitectura común de la familia de productos, y en COTS la actividad clave es comprar los componentes requeridos.

El Cuadro 3.4 muestra estas y otras comparaciones entre estos tres enfoques a lo largo de diferentes aspectos en las dimensiones organizacionales, técnicas y económicas.

Aspectos	CBSE	PLE	COTS *Based Development*
■ Organizacionales			
Interacciones principales	-Ingenieros de componentes y de aplicaciones	- Ingenieros de dominio	-Vendedor de componentes
Puestos principales (claves)	-Ingeniero de componentes -Ingeniero de aplicaciones -Encargado de la biblioteca	-Ingeniero de dominio -Arquitecto	-Administrador de reutilización.
■ Técnicos			
Integración	-Por composición (con otros componentes) -Generativa	-Por composición (en la arquitectura de dominio) -Generativa	-Por composición
Restricciones	-Los componentes se apegan a un modelo de componentes (no necesariamente a las restricciones del dominio)	-Los componentes se apegan a las restricciones de la arquitectura de dominio	-Los componentes se apegan a las restricciones de las aplicaciones
Encargados del soporte	-Ingenieros de aplicación	-Ingenieros de dominio	-Vendedor de componentes
Empaquetado	-Binario -El código fuente puede estar disponible para modificación	-Arquitectura común, infraestructura de comunicaciones, instrucciones, componentes binarios, código fuente	-Binario -Caja negra
Obtención	-Bibliotecas públicas o de dominio	-Bibliotecas del dominio	-Comercial
Tecnología de implementación	-Definida por el modelo de componentes	-Definida por la arquitectura común	-No controlable
■ Económicos			
Actividad de adquisición	-Adaptar	-Instanciar	-Comprar
Ámbito del mercado con la perspectiva del:	-Desarrollador de aplicaciones	-Ingeniero de dominio	-Proveedor de componentes
Motivaciones económicas	-Modelos de componentes comunes	-Diseño y arquitectura comunes.	-Mercado común

Cuadro 3.4 Comparación entre CBSE, PLE y COTS. (Mili, y otros, 2002)

Capítulo 4. Separación de intereses y orientación a aspectos

4.1. Separación de intereses

Hablar de la orientación a aspectos (OA) implica abordar obligadamente el problema de la separación de intereses, un tema que ha sido estudiado desde hace ya varias décadas desde diferentes perspectivas y que ha sido señalado como un principio fundamental de la ingeniería de software. La OA es una propuesta de solución relativamente reciente a este problema.

Un interés es una parte o unidad que es relevante para un concepto particular, objetivo o propósito relacionado con el software que se está desarrollando. En un sistema de software se involucran dos tipos de intereses: los que están relacionados con la funcionalidad básica, también llamada lógica del negocio de la aplicación, y con la funcionalidad secundaria, aquella que debe implementarse, pero que no forma parte de la lógica principal y generalmente "atraviesa" o compete a varias partes del sistema. La Orientación a Objetos (OO) permite enfrentar la funcionalidad básica de los sistemas de software. Sin embargo, como el paradigma obliga al desarrollador a elegir una descomposición dominante de intereses y apegarse a ella, los elementos de la funcionalidad secundaria quedan diseminados por varios módulos de la funcionalidad básica y no pueden ser encapsulados separadamente en un solo lugar.

Los intereses que tienen la característica de impactar (atravesar) a otros más reciben el nombre de transversales (*Crosscutting Concerns*). Los intereses relacionados con las funcionalidades secundarias son típicamente transversales; aunque no necesariamente un interés transversal debe ser asociado a funcionalidades secundarias. Algunos ejemplos de intereses transversales son: seguridad, manejo de errores, sincronización, persistencia de datos, manejo de memoria, acceso o *logging*, restricciones de tiempo, uso transparente de transacciones, conexiones a bases de datos, manejo de comunicaciones, etc. (ver Figura 4.1).

La descomposición de intereses de un sistema de software puede concebirse como un espacio n-dimensional, en donde cada interés del sistema ocupa su propia dimensión ortogonal al resto. Al momento de utilizar técnicas como la OO, que imponen una descomposición principal dominante y por lo tanto trabajan con un único espacio dimensional, se origina una mezcla no deseada de lógica principal con intereses transversales (ver Figura 4.2). En un sistema representado en un espacio unidimensional, los intereses transversales dan origen a intereses dispersos y enredados; los cuales introducen características indeseables como dificultad de reutilización, mantenimiento y evolución, calidad pobre y disminución de la productividad.

Los intereses dispersos (*scattered*) se presentan cuando un interés transversal se encuentra esparcido por diferentes partes del sistema. Se denominan intereses enredados (*tangled*) a los intereses que contienen funcionalidad principal y además incorporan competencias adicionales.

La OA aborda el manejo de los intereses dispersos mediante la incorporación de aspectos. Un aspecto puede definirse como un interés que atraviesa la aplicación, por lo que no puede encapsularse claramente, y que constituye una unidad conceptual susceptible de ser identificada y separada (ALGORRY, Marcelo, 2005). Por otra parte, Kiczales, uno de los creadores reconocidos de la POA, define un aspecto como una unidad modular diseminada por otras unidades funcionales y señala que los aspectos existen en la etapa de diseño así como en la de implementación.

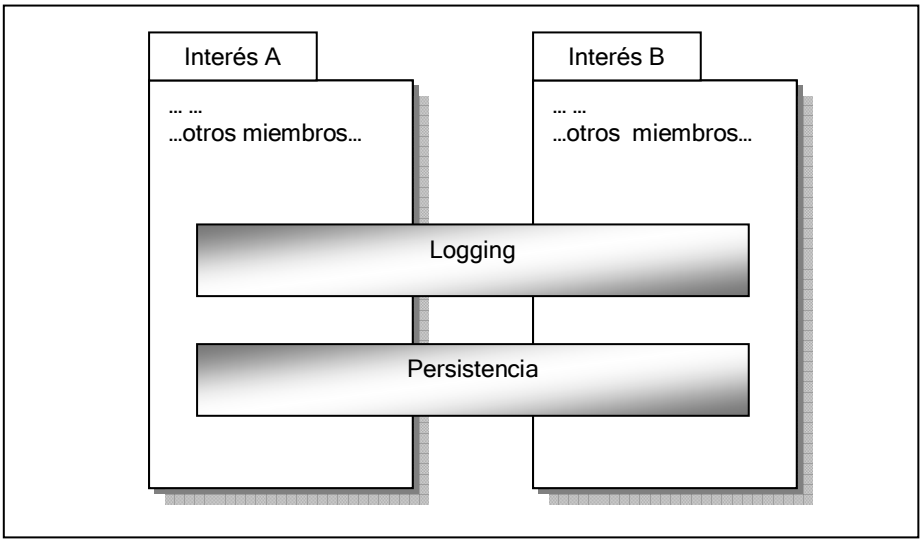


Figura 4.1. Intereses transversales (Crosscutting Concerns) en OO. (ALGORRY, Marcelo, 2005).

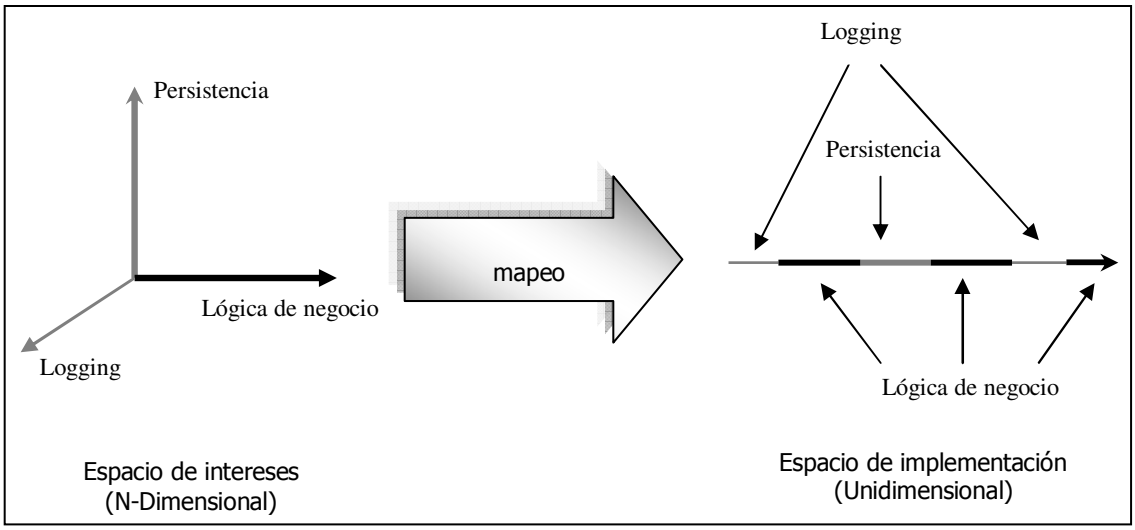


Figura 4.2. Separación de intereses y su implementación en un espacio unidimensional. (LADDAD, Ramnivas, 2003).

4.1.1. Breve evolución histórica de la separación de intereses

Dijkstra, en la década de los setenta introdujo el término "Separación de intereses" para referirse a la identificación, encapsulamiento y manipulación de partes del software que son relevantes para un concepto particular. También en los setenta, David Parnas (PARNAS, D. L., 1972) reconoció que la mejor manera de crear sistemas era identificando y separando los intereses del sistema utilizando la modularización.

Desde entonces la separación de intereses se ha convertido en un tema de atención central en la Ingeniería de software, y a través de los años, ha caracterizado a cada paradigma de desarrollo que ha surgido. La orientación a aspectos no es la excepción.

Las primeras ideas identificadas como orientadas a aspectos surgieron a principios de los noventa. Por ejemplo, para ese tiempo, el grupo Demeter trabajaba en la Programación adaptativa que posteriormente se consideró como la versión inicial de la POA. Luego, en 1993, William Harrison y Harold Ossher identificaron la necesidad de enfrentar los problemas de separación de intereses en un programa ya no desde la perspectiva de los objetos sino de un conjunto de vistas subjetivas de ellos; fue así como surgió la Programación Orientada a Sujetos (POS), un trabajo previo al modelo de separación multidimensional de intereses.

A mediados de los noventa, se perfilaba ya una concepción más madura de los aspectos. Por ejemplo, en 1995, Cristina Lopes y Walter Huersch presentaron un informe sobre la separación de conceptos con técnicas como Filtros de Composición y Programación Adaptativa para los intereses transversales, donde se abordó el tema de la separación de intereses como un problema muy importante por resolver en ingeniería del software. También en ese año, el grupo Demeter publicó la primera definición de aspecto. Un año más tarde (1996), Gregor Kickzales, de los laboratorios Xerox PARC (Palo Alto Research Center) introdujo el término de Programación Orientada a Aspectos, acotó su marco de trabajo y agregó precisión a la definición de un aspecto.

A finales de los noventa y durante el nuevo milenio, la orientación a aspectos fue atrapando cada vez más el interés de los desarrolladores y académicos, quienes poco a poco la han adoptado como tema central de sus trabajos. Por ejemplo, en 1999, Pari Tarr y Stanley M. Sutton Jr. haciendo énfasis en los problemas de trabajar con una sola dimensión de intereses impuesta por la orientación a objetos, subrayaron la llamada "tiranía de la descomposición dominante" y explicaron por primera vez un modelo multidimensional de separación de intereses con una perspectiva genérica que podía ser adoptada en diferentes etapas del ciclo de desarrollo.

Hoy en día, entre los investigadores más destacados en la evolución de la orientación a aspectos, por citar algunos, se encuentran Gregor Kickzales, Cristina Lopes y Karl Lieberherr, quienes han realizado aportaciones relevantes a la POA con el modelo bidimensional de separación de intereses. El equipo de trabajo de Kickzales es también

creador de Aspect-J, uno de los primeros lenguajes orientados a aspectos de propósito general basado en Java que fue liberado en 1997, y transferido a la comunidad de código abierto a finales del 2002. En cuanto al modelo multidimensional de separación de intereses aplicado a todas las fases de desarrollo, Tarr y Sutton están considerados entre los investigadores más relevantes por sus aportes y el trabajo significativo que han realizado en esta área.

4.2. Perspectivas para el estudio de la separación de intereses orientada a aspectos

Para hacer frente al problema de la descomposición dominante generado por los paradigmas como el orientado a objetos, han surgido varias soluciones alternativas, que permiten separar los intereses más limpiamente. En el nivel de implementación, en los últimos años ha ganado popularidad la Programación Orientada a Aspectos (POA o AOP – *Aspect Oriented Programming*- por sus siglas en inglés), un paradigma principalmente encabezado por Aspect-J (LADDAD, Ramnivas, 2003) que plantea la separación de intereses en dos dimensiones: la del modelo base y el aspectual. También se ha abordado la separación de intereses desde una perspectiva multidimensional (MDSOC, *Multi-Dimensional Separation Of Concerns* (TARR, Peri and Sutton, Jr., Stanley M., 1999)), que no impone el uso de un modelo base, y que si bien tiene herramientas de implementación, no es tan utilizado como el modelo propuesto por Aspect-J.

Los ejemplos de herramientas de programación típicos para ambos enfoques son: Aspect-J para el modelo bidimensional e Hiper-J (OSSHER, Harold and Tarr, Peri, 2000) para el multidimensional. Ambos lenguajes son extensiones de Java. También ambas perspectivas han sido abordadas desde las primeras etapas del ciclo de vida con la misma influencia conceptual de la etapa de implementación, por lo que se puede considerar a ambos enfoques como modelos genéricos que no son particulares a una sola fase del ciclo de vida. El modelo de OA que separa los aspectos de los componentes funcionales, y que implementa Aspect-J ha sido identificado atinadamente como bidimensional en algunos trabajos (CONSTANTINIDES, Constantinos A. et al., 1999) y (FERREIRA, Ricardo and Raminhos, Ricardo, 2005), aunque no es común encontrarlo en la bibliografía con esa nomenclatura.

En (HARRISON, William et al., 2002) se aborda el proceso de composición como una manera de construir software y se clasifican los paradigmas de los enfoques de composición de software en simétricos y asimétricos. Se identifica el enfoque utilizado por el modelo multidimensional (MDSOC) como simétrico, pues todos los componentes son bloques de construcción de estructura homogénea y no existe un modelo base al cual apegarse desde el inicio. Por otra parte, se identifica al enfoque utilizado por AspectJ como asimétrico, pues los aspectos son unidos a componentes de un modelo base; en este caso,

los aspectos y los componentes tienen estructuras distintas. Los componentes no pueden interactuar con otros componentes, ni los aspectos lo pueden hacer entre sí.

El concepto de paradigmas simétricos y asimétricos aplicados a procesos particulares de la Ingeniería del Software da lugar a distintos enfoques o metodologías específicas. De esta manera, por ejemplo, hablar de simetría resulta más general, mientras que hablar de separación multidimensional de intereses resulta más específico, pues éste último es un concepto más incluyente, que aunque está fundado sobre un paradigma simétrico, persigue además otros objetivos más particulares que se encuentran detallados en el apartado 4.2.2.

Por otra parte, si bien el término "orientado a aspectos" se utiliza comúnmente para referirse a una separación de intereses en un espacio bidimensional, en la categorización de enfoques de análisis y diseño propuesta en (CHITCHYAN, Ruzanna et al., 2005), el modelo de separación de intereses multidimensionales también se considera orientado a aspectos. En ese trabajo, se categoriza como enfoque de ingeniería de requerimientos orientada a aspectos toda aquella técnica que explícitamente considera el manejo limpio de intereses transversales (funcionales y no funcionales) y no-transversales; mientras que en la categoría de enfoques no orientados a aspectos se incluyen el resto de los trabajos contemporáneos que han sido fundamento importante para los nuevos enfoques orientados a aspectos.

De igual manera, en (ELRAD, Tzill and Filman, Robert E., 2001) se utiliza el término "Programación Orientada a Aspectos" para incluir programación adaptiva, filtros de composición, programación orientada a sujetos y separación multidimensional de intereses. Los autores señalan que utilizaron el término POA para englobar todos los enfoques que permiten expresar intereses transversales debido a la facilidad para recordarlo y utilizarlo sin ambigüedad.

En el presente trabajo también se tomarán estos criterios y se abordará el "modelo de separación aspectual bidimensional de intereses", para denotar la separación de componentes funcionales y aspectos, y el "modelo de separación aspectual multidimensional de intereses", para indicar que no hay una descomposición inicial obligada a la cual apegarse, persiguiendo los principios de MDSOC (TARR, Peri and Sutton, Jr., Stanley M., 1999) .

4.2.1. Modelo de separación aspectual bidimensional de intereses

Este modelo permite capturar los intereses transversales en entidades perfectamente definidas llamadas aspectos (ver Figura 4.3). De esta manera, los aspectos diseminados por el sistema se separan de la funcionalidad principal y cada uno de ellos se separa entre sí; así el desarrollador puede ver cada uno de esos aspectos como una entidad aislada con la cual puede trabajar más eficientemente.

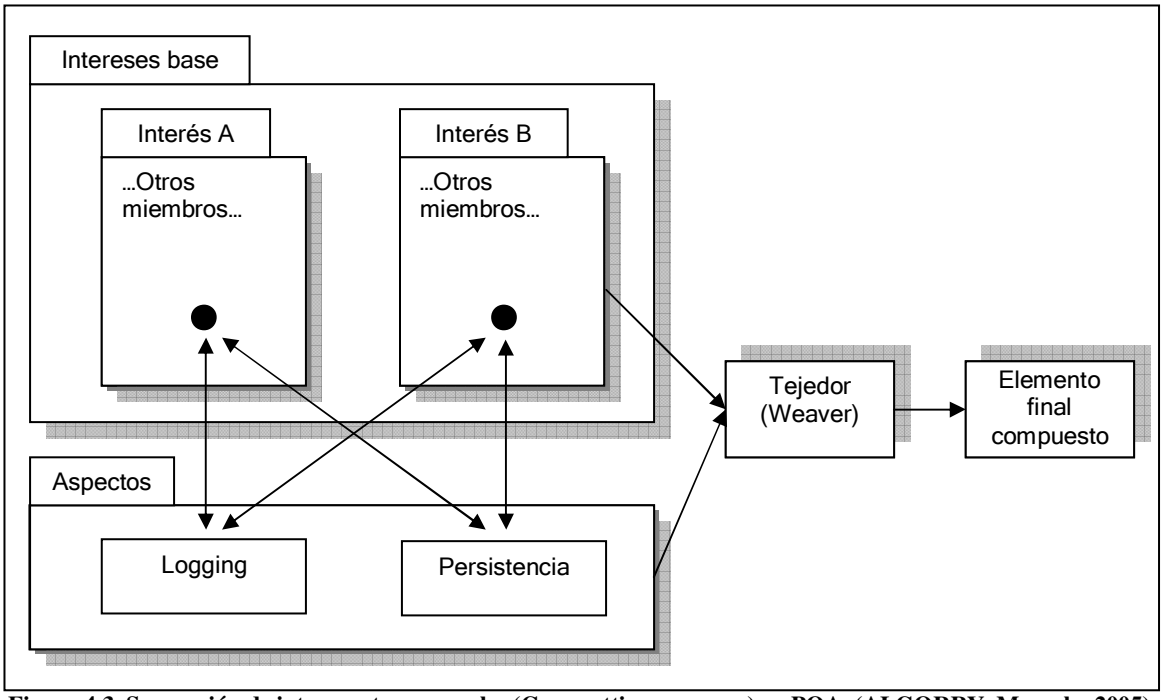


Figura 4.3. Separación de intereses transversales (Crosscutting concerns) en POA. (ALGORRY, Marcelo, 2005).

En el modelo bidimensional, los intereses transversales se encuentran encapsulados en los aspectos, éstos están siempre relacionados con los intereses base y por lo tanto, deben ser compatibles con ellos. La interacción entre los intereses de funcionalidad base y los aspectos se realiza mediante puntos de enlace (*join points*), los cuales son lugares específicos, bien identificados en los intereses base donde se realiza la modificación con el comportamiento adicional descrito en los aspectos. La descripción de los puntos de enlace se encuentra autocontenida en el aspecto junto con la descripción del interés transversal.

Los puntos de enlace de los aspectos pueden hacer referencia a varios lugares de los intereses base en donde actuará el aspecto sin tener que enumerarlos uno por uno. A esta propiedad se le denomina cuantificación (KICILLOF, Nicolás, 2008). La cuantificación más común es la que permite precisar lugares de acuerdo a los identificadores que se encuentran presentes en los intereses base.

Los intereses de la funcionalidad base no deben conocer los detalles de los aspectos, ni deben estar al tanto de su existencia. A esta característica se le denomina prescindencia. Los aspectos, por el contrario, deben conocer con detalle suficiente los intereses de la funcionalidad base pues actúan siempre sobre ellos. Esta relación estrecha siempre existirá entre aspectos e intereses base.

El tejedor o entrelazador (*weaver*) es el encargado de combinar los módulos de la funcionalidad básica con los aspectos para generar un producto final compuesto. Para realizar este proceso conocido como entretejido, integración o composición, el tejedor se guía por los puntos de enlace definidos dentro de los aspectos.

La Figura 4.4 muestra la comparación de la estructura típica de un sistema de software con funcionalidad dispersa implementado con OO y OA con un modelo bidimensional. Es importante notar que la estructura orientada a aspectos permite analizar más limpiamente cada interés; esto facilita el mantenimiento de cada interés por separado y del sistema en general.

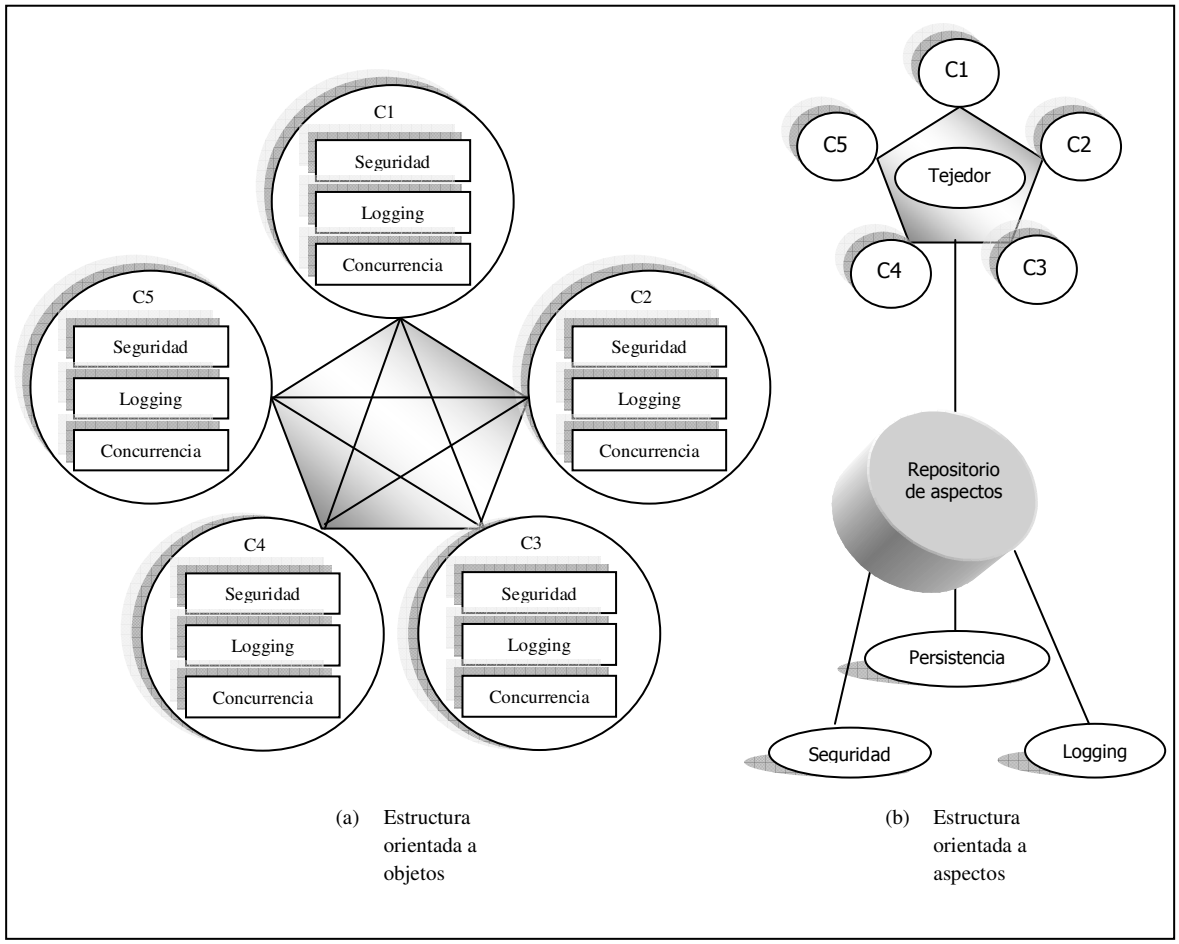


Figura 4.4. Comparación de la estructura de un programa con OO y OA. (ELRAD, Tzilla et al., 2004).

4.2.2. Modelo de separación aspectual multidimensional de intereses

Contrario al modelo aspectual bidimensional, el modelo multidimensional considera que todos los intereses tienen la misma importancia, y no requiere que se lleve a cabo una descomposición inicial que los separe en categorías de funcionalidad básica o secundaria. Al liberar esta restricción a la que se apega el modelo bidimensional, se obtiene mayor flexibilidad para organizar los intereses y realizar combinaciones entre ellos para lograr

nuevas configuraciones sin tener que hacer cambios drásticos que impacten en la estructura global del sistema.

La principal motivación de este modelo es superar la limitación de la llamada "tiranía de la descomposición dominante" a la cual se apegan gran parte de las metodologías, formalismos y herramientas existentes. La tiranía se refiere a la restricción de realizar la separación y encapsulación de un solo tipo de intereses de a la vez.

El modelo multidimensional fue planteado por Tarr y Sutton en 1999; en un trabajo posterior (OSSHER, Harold and Tarr, Peri, 2000) Tarr explica: "*El concepto de separación multidimensional de intereses, denota una separación de intereses que involucra:*

- *Múltiples y arbitrarias dimensiones de interés.*
- *Separación simultánea a lo largo de esas dimensiones.*
- *La habilidad de manejar nuevos intereses y nuevas dimensiones de interés dinámicamente, conforme van surgiendo a través del ciclo de vida.*
- *Intereses traslapados y con interacción entre ellos; es atractivo pensar que los intereses son independientes entre sí u "ortogonales" pero raramente lo son en la práctica*."

Una dimensión de interés es una manera de descomponer, organizar y estructurar el software de acuerdo a intereses de un tipo particular. Manejar varias dimensiones simultáneamente implica tener varias formas de trabajar con los distintos intereses del software al mismo tiempo; y así poder configurar fácilmente la estructura del sistema no solo de una sino de varias maneras.

Lograr tener varias descomposiciones de intereses con los paradigmas unidimensionales como OO implica realizar cambios invasivos, planeación adicional, y reingeniería, lo que se traduce en un aumento considerable del tiempo de desarrollo, del costo del proyecto y de la complejidad del software.

Aunque el modelo bidimensional propone una separación de los intereses transversales mediante el uso de aspectos, éstos últimos mantienen un alto grado de acoplamiento con los intereses base, lo que dificulta su reutilización: si la estructura de los intereses base cambia y afecta los puntos de enlace indicados por los aspectos, éstos también deben modificar su estructura para reflejar esos cambios. También los aspectos requieren de los intereses base para completar su significado. Además, como los aspectos trabajan sobre la descomposición dominante elegida para los intereses base, no aportan elementos para superar la rigidez estructural que ésta provoca. De esta manera, el grado de flexibilidad de trabajar con los intereses base también es una limitante.

Al tratar a todos los intereses de manera homogénea, el modelo multidimensional logra una separación de intereses más aislada que el modelo bidimensional. También proporciona mayor flexibilidad de trabajo y capacidad de reutilización al permitir que el desarrollador maneje diferentes descomposiciones para un mismo conjunto de intereses (ver Figura 4.5).

Con esto se supera el acoplamiento entre aspectos e intereses base del modelo bidimensional.

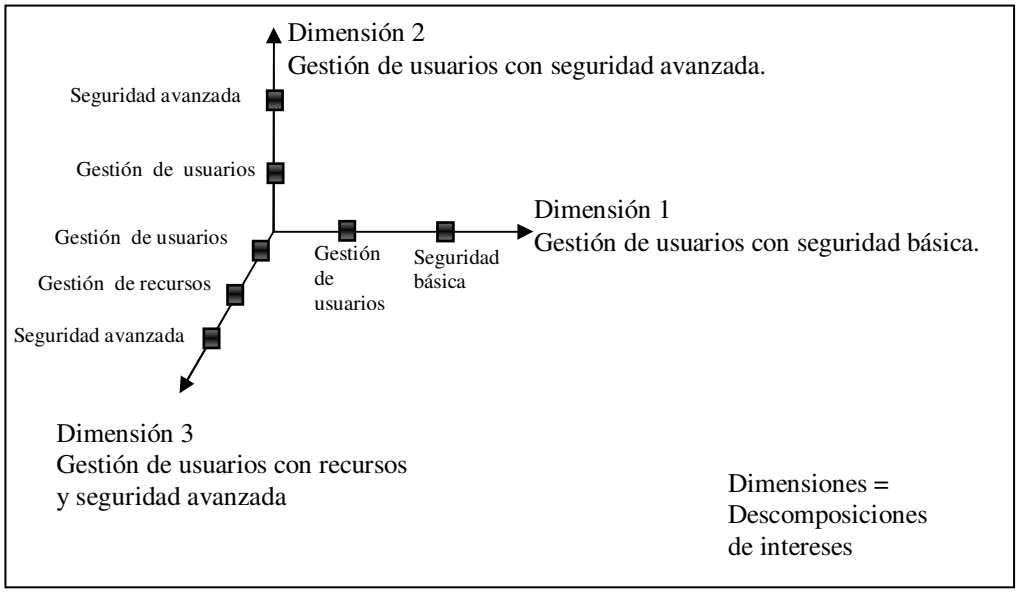


Figura 4.5. El modelo de separación de intereses multidimensional.

El modelo multidimensional abstracto propuesto por Tarr y Sutton no especifica las unidades de trabajo ni detalles particulares de implementación, por el contrario deliberadamente los dejan abiertos y libres para que cada enfoque los utilice y los incorpore a su manera de trabajo en las diferentes fases del ciclo de vida. Los elementos más importantes de su modelo son:

- Unidad.- Es una construcción sintáctica, por ejemplo, una declaración, una sentencia, un diagrama de estados, una clase, una especificación de requerimientos, etc.
- Espacio de interés.- Abarca todas las unidades en algún cuerpo del software, por ejemplo, un conjunto de sistemas de software, o librerías o una familia de productos.
- Hiper espacio.- Es un espacio de interés especial para soportar la separación de intereses multidimensionales. Consiste en una matriz multidimensional en donde cada eje representa una dimensión de interés (forma de descomposición) y cada punto en un eje es un interés en esa dimensión. Los intereses en cada eje no pueden traslaparse, pero los intereses en diferentes dimensiones sí pueden hacerlo.
- Hiper plano.- Es el conjunto de unidades que afectan a un interés. Cada interés en una dimensión define un hiper plano.
- Hiper sección.- Es un conjunto de unidades que están declarativamente completos, es decir, ninguna hiper sección depende de otra; si ésta llegase a necesitar un elemento que no está definido dentro de ella, utiliza declaraciones abstractas que en la integración se completan. Una hiper sección es un bloque de construcción que

puede ser integrado para formar otros bloques más grandes. Cualquier conjunto arbitrario de unidades puede convertirse en una hiper sección.

- Hiper módulo.- Comprende un conjunto de hiper sección y un conjunto de relaciones de integración que especifican la manera en la que las hiper secciones se relacionan una con otra y como se deben integrar.

Las unidades, hiper secciones e hiper módulos son elementos importantes en el modelo multidimensional. Para mostrar la instanciación del modelo con un ejemplo de alto nivel y una comparación gráfica con el modelo bidimensional, se analiza a continuación la Figura 4.6.

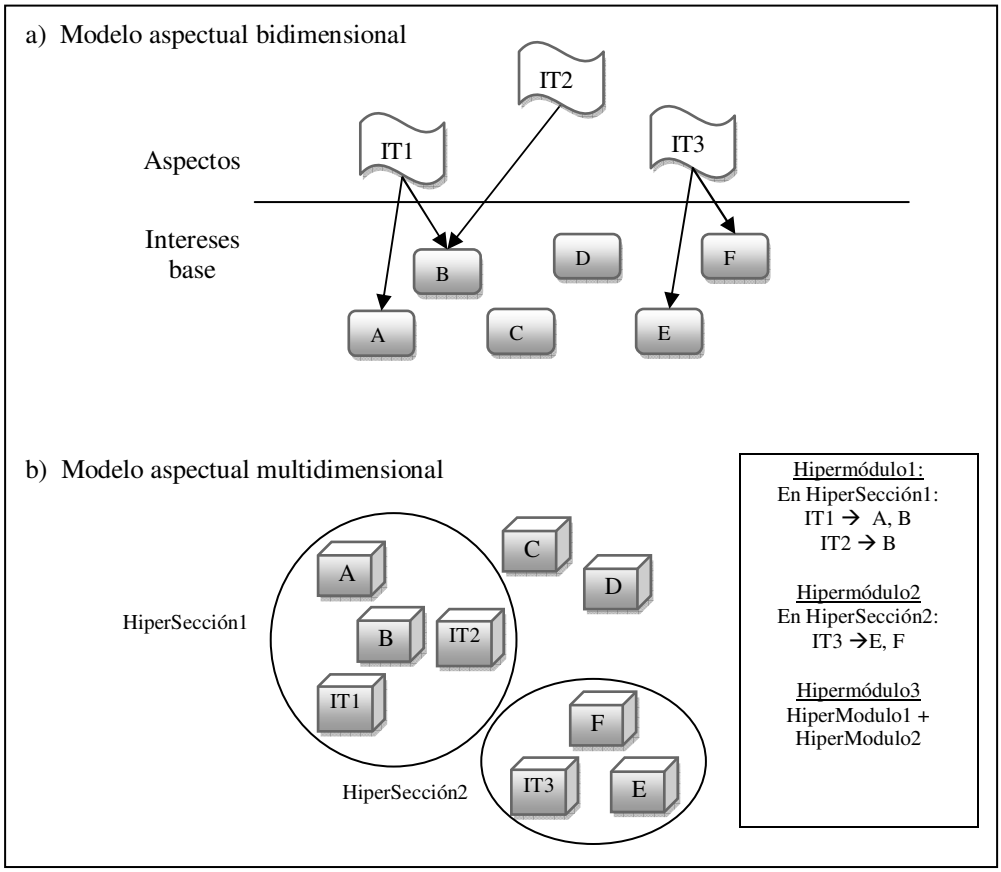


Figura 4.6. Comparación de modelos bidimensional y multidimensional.

En la Figura 4.6(a) se observa la separación inicial entre intereses base y aspectos que el desarrollador debe definir al trabajar con el modelo bidimensional. Los aspectos impactan a los intereses base y entre ellos no se pueden relacionar. La Figura 4.6(b) es una instanciación particular del modelo genérico multidimensional que sigue sus principios

básicos a un alto nivel para representar la misma situación planteada en la Figura 4.6(a) tomando en cuenta el enfoque particular del presente trabajo de tesis para manejar intereses.

Nótese en la Figura 4.6(b) el uso de hiper secciones para agrupar intereses según se requiera, y el uso de hiper módulos para indicar las relaciones entre los intereses separadamente de su estructura individual; es así como los intereses no afectan su contenido ante los cambios de los otros intereses con los cuales se relacionan ni ante los cambios en las relaciones que se den entre ellos mismos.

Los hiper módulos pueden ser anidados, por ejemplo, *Hipermódulo3* especifica una composición global para el sistema, en base a otros dos hiper módulos definidos previamente que especifican solo parte del sistema completo.

El modelo multidimensional mediante el uso adecuado de hiper secciones e hiper módulos facilita la arquitectura de sistemas cuya configuración involucre características opcionales o de selección (*mix-and-match*). Sin embargo, el buen juicio del desarrollador siempre será imprescindible para lograr un equilibrio entre las posibilidades que permite el modelo y el grado de complejidad y buen diseño que se desee obtener.

La gran aceptación del uso de aspectos en la implementación, predominantemente encabezado por Aspect-J, ha impactado a las primeras actividades del ciclo de desarrollo, por lo que, por ejemplo, es más común encontrar enfoques de especificación de requerimientos con influencia del modelo bidimensional que con separación multidimensional de intereses.

4.3. Beneficios de la separación de intereses orientada a aspectos

Los principales beneficios que aporta la OA en general son derivados de su manera de modularizar conceptos que permite reducir la complejidad, mejorar la reutilización, y facilitar la evolución; todo esto se traduce en una mejoría global de la calidad del software.

- La separación de intereses permite que los desarrolladores trabajen de manera más rápida e independiente, con lo que aumenta la productividad del equipo de desarrollo. Al contribuir con un mejor uso los recursos, la OA reduce costos en el desarrollo actual y en las futuras modificaciones que se realicen al sistema.
- La orientación a aspectos facilita la abstracción y reduce las dependencias entre los módulos, asignándoles responsabilidades individuales. Como consecuencia, es posible construir aplicaciones cuyo ciclo de vida resulta más sencillo de abordar: los sistemas creados evolucionan mejor y aceptan la inclusión de nuevas características, su construcción y pruebas son más fáciles de realizar, y su desempeño no se ve alterado por la introducción de este paradigma.

- También se eliminan duplicidades, se superan los problemas causados por intereses dispersos y enredados y se promueve una organización más limpia, ordenada, fácil de mantener, reutilizar y refactorizar.
- La OA desde un punto de vista sistemático promete una trazabilidad directa desde los requerimientos y diseño hacia la implementación y prueba, tal como existe para la orientación a objetos desde hace algunos años.

Dentro de las actividades que se pueden ver beneficiadas al incluir el uso de las técnicas orientadas a aspectos se encuentran: el análisis de escenarios no-invasivos del tipo "*qué pasaría si*", la simulación de escenarios complejos (agregando fallas al sistema para probar su respuesta), y el soporte de traza en los reportes de errores o *logging* no-invasivo para comprender el comportamiento del sistema (LADDAD, Ramnivas, 2003).

4.4. Problemas de la separación de intereses orientada a aspectos

La OA presenta algunos inconvenientes, uno de ellos es que es un paradigma relativamente nuevo para algunos los desarrolladores y deben familiarizarse bien con él antes de lograr una buena productividad. La marcada curva de aprendizaje inicial, especialmente en proyectos a gran escala, puede considerarse un inconveniente. Sin embargo, una vez asimilados y puestos en práctica, los conocimientos de OA son fácilmente transferibles a otros proyectos, reduciendo drásticamente la curva posterior. Otro punto argumentado en contra de la OA es que requiere el uso de buenas herramientas para la administración de intereses y llevar a cabo eficientemente su proceso de entretejido o integración. Sin embargo, las técnicas y herramientas orientadas a aspectos no son lo suficientemente maduras ni comercializadas, especialmente en las primeras fases del ciclo de vida. Las herramientas más estables se encuentran disponibles para la fase de implementación.

Actualmente el ciclo de vida orientado a aspectos en general carece de actividades y herramientas totalmente unificadas y aceptadas. Por ejemplo, el análisis y el diseño orientado a aspectos necesitan de una representación estándar equivalente al UML adoptado para la orientación a objetos, pues aunque algunas propuestas extienden de diferentes maneras el UML para representar aspectos, hasta el momento se consideran solamente como tendencias y opciones de modelado.

Los detractores también dicen que con OA se pueden introducir efectos secundarios desconocidos al realizar un cambio en el sistema, tales como "choques" o configuraciones inválidas. Los "choques" representan conflictos de nombres o de situaciones que culminan en resultados no esperados. Comúnmente se observan cuando más de un interés actúa sobre otro. Estos escenarios son aún más probables al aumentar el número de intereses con espacio de trabajo traslapado. Las configuraciones inválidas en la estructura del software se pueden obtener principalmente al relacionar intereses separados con un enfoque multidimensional.

Otro argumento en contra de las técnicas de OA, es que dejan abierta la posibilidad de introducir comportamiento malicioso y furtivo en los sistemas si no existe un adecuado esquema de seguridad en el entorno de desarrollo. La realidad es que efectivamente, tanto choques como configuraciones inválidas y comportamientos maliciosos son posibles con OA, pero el antídoto efectivo consiste en utilizar este paradigma con mesura, precaución y de manera adecuada y controlada, tal como ocurre con cualquier otro paradigma. Aplicada de esta forma, la OA ofrece siempre más beneficios que complicaciones.

Por otra parte, algunos desarrolladores también pueden debatir que la depuración de errores en un sistema orientado a aspectos resulta más difícil al presentarse alteraciones en el flujo de la secuencia lógica como consecuencia de la separación de intereses; sin embargo, las herramientas y entornos de trabajo son los encargados de proveer mecanismos automatizados que hagan que la tarea de depuración no resulte más compleja que en un sistema puramente orientado a objetos.

4.5. Conclusiones

Aunque el nacimiento de la OA es relativamente reciente como tal, su comunidad de adeptos, principalmente en la etapa de programación, ha crecido mucho en los últimos tiempos. En la actualidad, la OA es un paradigma que continúa evolucionando y no tiene completamente definidos ni formalizados sus límites y posibilidades; sin embargo, por su importancia, con seguridad será tan utilizado como lo es hoy en día la orientación a objetos.

Mientras tanto, el ambiente empresarial aún no adopta el paradigma con la seriedad debida; tal vez por su reciente aparición, o por carecer de técnicas estandarizadas adecuadas que soporten las actividades del ciclo de vida completo y garanticen una buena productividad en proyectos a gran escala. Actualmente ha crecido el número de opciones de implementación con OA en sistemas de propósito general. Sin duda Aspect-J se ha convertido en el estándar para POA y su impacto se ha comparado con el de Smalltalk para la POO (FIGUEROA, Pablo and Isaza, Pablo, 2006).

La OA ha capturado la atención de los investigadores y se ha ubicado como un tema de actualidad en la comunidad académica, principalmente con la búsqueda de propuestas para incorporar una correcta separación aspectual de intereses en las primeras actividades de desarrollo de software. Sin embargo, aunque algunas instituciones educativas vanguardistas han comenzado a ofrecer cursos acerca de POA comúnmente como asignaturas opcionales, conferencias de actualización, seminarios o talleres, aún no se ha incorporado hasta el momento una materia formal de POA en los planes de estudio de programación en las carreras de informática. Por otra parte, el número de libros publicados de OA es escaso comparado con los que abordan la orientación a objetos. Sin embargo, las editoriales ya ofrecen diferentes títulos referentes a la orientación a aspectos, la mayoría de

programación OA desde la perspectiva de Aspect-J y se han convertido en una excelente fuente de referencia.

También hoy en día se intenta incorporar de diversas maneras el uso homogéneo de aspectos en todas las etapas de desarrollo de software. Actualmente existen ya ciertas iniciativas de unificación. Además, han surgido guías de diseño orientado a aspectos que describen usos incorrectos de la POA, patrones, y soluciones orientadas a aspectos para problemas específicos.

Capítulo 5.
Aspectos tempranos

5.1. Introducción

El término "aspecto temprano" (*early aspect*) se puede referir a un interés que atraviesa la descomposición dominante de un elemento, o a un módulo base derivado de la separación dominante de intereses. En ambos casos el término "temprano" ubica al aspecto en las etapas del ciclo de vida de desarrollo de software previas a la implementación (BANIASSAD, Elisa et al., 2006), es decir, durante el análisis de dominio, en la etapa de la ingeniería de requerimientos, durante el diseño de arquitectura y cuando se realiza el diseño del sistema.

Identificar y manejar los aspectos desde estas etapas tempranas ha mostrado algunas ventajas en la práctica. Algunas de las más relevantes consisten en mejorar la modularidad de los sistemas, lograr una mayor consistencia entre requerimientos y diseño de arquitectura e implementación, dar seguimiento claro a los aspectos a lo largo del ciclo de vida, asegurarse que los intereses transversales en el dominio de un problema o en un espacio de solución sean capturados como aspectos en la implementación, y detectar conflictos en los intereses antes de que sea demasiado costoso resolverlos en etapas avanzadas.

Para manejar los aspectos desde las primeras actividades del ciclo de vida se debe aplicar alguna técnica o enfoque; las opciones varían en función de diferentes criterios, por ejemplo, la etapa en la que se aplican, el modelo de separación de intereses que usan, el grado de formalidad que incorporan, etc. Sin embargo, el objetivo general de todas las técnicas de desarrollo orientadas a aspectos es "*proporcionar medios para la identificación sistemática, la modularización y composición de intereses transversales durante todo el ciclo de vida del software*" (BLAIR, G. et al., 2004).

Hoy en día, las primeras etapas de desarrollo de software han tomado conceptos provenientes de los modelos de implementación existentes en la programación orientada a aspectos como Aspect-J e Hyper-J y han dado lugar a múltiples técnicas que pueden ser utilizadas en conjunto con otras durante cada etapa. La elección de las técnicas particulares debe ser el resultado de un análisis minucioso de los factores que intervienen en el proceso, tales como los requerimientos del sistema, las prácticas de la organización, las restricciones, y la naturaleza de los intereses transversales que se van a considerar.

La orientación a aspectos ha comenzado a ser muy utilizada en la fase de programación; sin embargo, el modelado de los intereses no se realiza aún de manera independiente y uniforme en todo el ciclo de vida. En el desarrollo de software orientado a aspectos se busca que los intereses sean siempre entidades identificables que puedan modelarse en sus propios formalismos separadamente en cada etapa del proceso (SUTTON, Stanley Jr. and Rouvellou, Isabelle, 2004).

De acuerdo a las mejores prácticas de la Ingeniería del Software, se han identificado algunas características importantes deseadas en las técnicas de desarrollo orientadas a aspectos (CHITCHYAN, Ruzanna et al., 2005); estas características, que se muestran a continuación, han servido incluso como criterios básicos de comparación entre las diversas técnicas existentes:

- Trazabilidad.- Permite el entendimiento de los requerimientos y sus productos posteriores asociados. Se pueden identificar dos tipos:
    - Trazabilidad de los requerimientos y cambios hacia su origen.- Se refiere a la capacidad de identificar de donde proviene un elemento, o algún cambio que se le haya realizado a él.
    - Trazabilidad de los requerimientos hacia su próxima representación en el ciclo de vida.- Cuando un elemento evoluciona a una etapa posterior del ciclo de vida se debe tener un enlace que describa la relación entre ambas representaciones.
- Composición.- Es la habilidad de tomar elementos, conjuntarlos de acuerdo a reglas y ver al sistema como un todo integrado por elementos compuestos.
- Evolución. – Se refiere a la facilidad para modificar, eliminar o agregar elementos sin demasiado esfuerzo por parte del desarrollador.
- Escalabilidad.- Debe ser igualmente útil para proyectos pequeños y grandes. Además debe apoyarse con herramientas computacionales.

En los siguientes apartados se aborda el uso de aspectos en las etapas de desarrollo previas a la implementación: el análisis de dominio (apartado 5.2), la ingeniería de requerimientos (apartado 5.3), la arquitectura (apartado 5.4) y el diseño del sistema (apartado 5.5).

5.2. Aspectos en el análisis de dominio

El análisis de dominio es el proceso de identificar, capturar y organizar conocimiento proveniente de un dominio con el objetivo de hacerlo reutilizable al crear nuevos sistemas. Existen dos razones para incorporar el análisis de dominio en el desarrollo orientado a aspectos: en primer lugar, es posible que además de los aspectos identificados durante la etapa de requerimientos, existan otros que se hayan pasado por alto, ya que el análisis de requerimientos se realiza en el dominio del problema y no en el dominio de la solución; se define éste último como el dominio que incluye la solución para los requerimientos.

La segunda razón es la importancia de entender perfectamente la estructura y el comportamiento de los aspectos previamente a la etapa de implementación. Como los modelos de dominio se consideran representaciones lo suficientemente estables para implementar intereses, un modelo de dominio con aspectos sería de mucha ayuda para comprenderlos e implementarlos correctamente.

El análisis de dominio orientado a aspectos es el área menos explorada dentro de los aspectos tempranos. Actualmente continúan en proceso algunos trabajos que intentan realizar propuestas para incorporar aspectos en la ingeniería de dominio. Mientras tanto, aún se desconocen con exactitud los indicadores más característicos para los enfoques emergentes. Por ejemplo, aún existe debate acerca de si un método de análisis de dominio orientado a aspectos debería diferenciar explícitamente los aspectos, o realizar algún tipo de consideración especial con ellos.

5.3. Aspectos en la ingeniería de requerimientos

La ingeniería de requerimientos es la fase del ciclo de vida que se ocupa de razonar acerca del dominio del problema y formular una especificación de requerimientos que refleje adecuadamente las necesidades del cliente. La especificación obtenida es un puente entre el dominio del problema y el dominio de la solución.

Un requerimiento es definido por el estándar IEEE 610.12-1990 (CLELAND-HUANG, Jane, 2005) de la siguiente manera:

- Es una condición o capacidad necesitada por un usuario para resolver un problema o alcanzar un objetivo.
- Es una condición o capacidad que debe ser alcanzada o poseída por un componente de un sistema para satisfacer un contrato, estándar, especificación u otros documentos formalmente establecidos.
- Es una representación documentada de una condición o capacidad como se describe en los dos puntos anteriores.

Una clasificación bastante común en la literatura es la que categoriza a los requerimientos de un producto de software como funcionales y no funcionales. Los requerimientos funcionales son las características principales que el sistema debe ofrecer. Los no funcionales describen restricciones acerca de las cualidades que se deben observar en el sistema. Los requerimientos no funcionales también reciben el nombre de requerimientos de calidad o sistémicos, pues pueden ser refinados progresivamente en requerimientos funcionales.

Para ser implementados, los requerimientos no funcionales dependen de los otros requerimientos a los cuales afectan, y de las decisiones de arquitectura o diseño que hagan posible su realización; por ejemplo, el tiempo de respuesta en una aplicación depende de cada una de las operaciones que se integran en el sistema y de la arquitectura utilizada en cada una de ellas.

La principal misión del proceso de desarrollo de software es construir un sistema confiable, eficiente, de calidad y económico a partir de un conjunto de requerimientos funcionales y no funcionales debidamente especificados. Es importante mantener esto en mente pues el

costo de corregir errores en los requerimientos se incrementa entre más avanzadas se encuentren las actividades en el ciclo de vida.

Mantener bien separados todos los intereses de importancia en un sistema de software contribuye grandemente a tener éxito en el desarrollo del sistema (TARR, Peri and Sutton, Jr., Stanley M., 1999). Para lograr un sistema modular y con intereses limpiamente separados, se prefiere que cada requerimiento sea implementado en un respectivo componente de software. Sin embargo, hay algunos requerimientos que no encajan en este esquema por lo que reciben el nombre de transversales o aspectuales. Los requerimientos no funcionales como por ejemplo, desempeño, seguridad, movilidad, y disponibilidad, son típicamente transversales pues tienen amplio impacto en otros requerimientos.

Todos los requerimientos, pero en especial los transversales, se pueden administrar mejor con una adecuada modularización. El objetivo es poder estudiarlos de manera individual, pero también en su relación con el resto; de esta manera se facilita la evolución y la trazabilidad de todos los requerimientos durante las fases del ciclo de vida.

Además de ser modular, cada requerimiento individual debe exhibir un conjunto de características que son necesarias para evitar la introducción de errores en la especificación (CLELAND-HUANG, Jane, 2005):

- Conciso.- Debe describir una sola característica del sistema y no debe incluir más información de la necesaria asociada a esa característica.
- Correcto.- Debe describir de forma precisa la característica del sistema, sin que falte información.
- No ambiguo.- Debe ser claro, entendible y sin lugar a múltiples interpretaciones.
- Factible.- Debe ser técnica, financiera y administrativamente factible.
- Verificable.- Debe proveer criterios claros de aceptación que permitan verificarlo y probarlo.

El documento que contiene la especificación de todos los requerimientos debe tener también algunas características deseables (FAIRLEY, Richard, 1994) y (LAMSWEERDE, Axel Van, 2000). En resumen, se desea que una especificación sea:

- Adecuada.- Debe abordar el problema de la manera más precisa posible, apegándose a su dominio en todo momento.
- Completa.- Las propiedades especificadas deben ser suficientes para describir todos los requerimientos en diferentes niveles de abstracción.
- Consistente.- Los diferentes requisitos deben estar especificados sin contradicción entre ellos. Es decir, debe haber una interpretación de los requisitos que haga verdaderas todas las propiedades especificadas.
- No ambigua.- Los requerimientos deben estar expresados de manera que solo una interpretación de interés esté relacionada con ellos.
- Funcional.- Los requerimientos deben expresar lo que el cliente necesita sin detallar “cómo” el sistema lo va a realizar.

- Mínima.- No se deben incluir detalles irrelevantes para el problema, o de importancia para una sola solución particular.
- Verificable.- Debe ser posible asegurar que los requerimientos reflejan las necesidades del cliente y que los productos de cada fase satisfacen esos requerimientos.
- Rastreable.- Debe ser posible rastrear cada requisito especificado hasta sus enunciados particulares; de esta manera se puede establecer una conexión clara entre los diferentes puntos del documento.
- Fácilmente modificable.- Debido a la naturaleza cambiante de las necesidades del cliente, debe ser posible introducir cambios sin tener que comenzar el proceso desde cero.

Las técnicas de ingeniería de requerimientos orientadas a aspectos (*Aspect-Oriented Requirements Engineering*) tienen como objetivo proporcionar medios para identificar los requerimientos funcionales y no funcionales, y modularizarlos adecuadamente para manejar los conflictos que surjan en su representación. Independientemente de la técnica elegida, siempre se desean obtener las características descritas anteriormente para cada requerimiento individual y para el documento de especificación.

El estudio de la ingeniería de requerimientos orientada aspectos ha sido muy importante y fructífero en los últimos años. Algunas de sus contribuciones más sobresalientes se resumen a continuación (CHITCHYAN, Ruzanna et al., 2005):

- Ha puesto en relieve la necesidad de tratar de igual manera a los intereses funcionales y no funcionales.

- Se ha descubierto que tanto requerimientos funcionales como no funcionales pueden tener una amplia influencia transversal sobre otros requerimientos.

- Se ha establecido el principio de separación temprana de intereses, que reconoce que debe ser posible estudiar cada requerimiento de manera separada.

- Se ha identificado la necesidad de un mecanismo de composición de requerimientos, con el cual debe ser posible integrar los requerimientos con otros para entender las interacciones entre ellos.

- Ha reconocido la necesidad de tener un soporte para tomar decisiones acerca de los intereses, aunado a un sistema de trazabilidad sistemática y conveniente para llevar un registro de las decisiones tomadas.

Existen diferentes enfoques para manejar la separación de intereses en la etapa de ingeniería de requerimientos. Algunos son métodos tradicionales que si bien no incorporan el concepto de aspecto explícitamente, permiten especificar y organizar los diferentes

intereses de acuerdo a sus criterios propios. Algunos otros, soportan explícitamente la orientación a aspectos. Una clasificación muy completa de los enfoques para ingeniería de requerimientos es la propuesta en (CHITCHYAN, Ruzanna et al., 2005), que se muestra en el Cuadro 5.1; las principales características de cada uno de estos enfoques se encuentran resumidas en el Cuadro 5.2.

El objetivo de la aplicación de cada uno de los enfoques de trabajo se alinea con la meta de la ingeniería de requerimientos: capturar requerimientos que sean correctos, completos y factibles; para lo cual, los requerimientos en todos los casos deben ser descubiertos, entendidos, especificados, verificados, comunicados y administrados por el desarrollador con ayuda de las herramientas provistas por los enfoques. En resumen, cada enfoque debe ayudar a que el desarrollador logre especificaciones de buena calidad.

- Enfoques no orientados a aspectos
  - Orientados a Puntos de vista
    - PREview
    - Viewpoints and Inconsistency Management (VIM)
  - Orientados a Metas
    - Non-functional Requirements Framework (NFRF)
    - Knowledge Acquisition in Automated Specification (KAOS)
    - I*
  - Orientados a Marcos Problema
    - Método de marcos problema
  - Orientados a Casos de uso y basados en escenarios
    - Casos de uso
    - Escenarios negativos y Casos de no-uso
- Orientados a aspectos
  - Basados en Puntos de vista
    - AORE con Arcade
  - Basados en metas
    - Aspects in Requirements Goal Models (ARGM)
  - Orientados a casos de uso y basados en escenarios
    - Aspect-Oriented Software Development with use cases (AOSD/UC)
    - Scenario Modeling with Aspects (SMA)
    - Aspectual Use Case Driven Approach (AUCDA)
  - Orientados a la separación multidimensional de intereses
    - Concern Modeling with Cosmos
    - Concern Oriented Requirements Engineering (CORE)
  - Basados en componentes
    - Aspect-Oriented Requirements Engineering for Component-Based Software Systems (AOREC)
  - Otros
    - Theme/Doc

Cuadro 5.1 Clasificación de los enfoques para ingeniería de requerimientos con separación de intereses. (Chitchyan, y otros, 2005)

Enfoque	1	2	3	4	5	6	7	8	9
PREview	✓	✓	×	×	×	✓	✓	n/a	×
VIM	✓	✓	✓	✓	✓	✓	✓	n/a	✓
NFRF	✓	✓	i	✓	✓	✓	✓	n/a	✓
Marcos problema	I	✓	✓	✓	✓	✓	×	n/a	✓
KAOS	✓	✓	i	✓	✓	✓	✓	✓	✓
I*	✓	✓	✓	✓	i	✓	✓	✓	✓
Casos de Uso	✓	✓	✓	✓	✓	×	✓	✓	✓
Casos de desuso	✓	✓	✓	✓	✓	×	✓	✓	✓
AORE/Arcade	✓	✓	✓	✓	✓	✓	✓	✓	✓
ARGM	✓	✓	✓	✓	✓	✓	✓	✓	✓
AOSD/UC	✓	✓	✓	✓	✓	✓	✓	✓	✓
SMA	✓	×	✓	✓	✓	×	×	✓	✓
AUCDA	✓	✓	✓	✓	✓	×	✓	✓	✓
Cosmos	×	✓	✓	✓	✓	✓	✓	✓	✓
CORE	✓	✓	✓	✓	✓	✓	✓	✓	✓
AOREC	×	✓	✓	✓	✓	✓	✓	n/a	✓
Theme/Doc	×	✓	✓	✓	✓	×	✓	✓	✓

- Notación utilizada:
  - × = No se considera
  - i = Indirectamente
  - ✓ = Sí se considera
  - n/a = No aplica

- Características de los enfoques
  1. Trazabilidad de los requerimientos y cambio hacia su origen.
  2. Trazabilidad de los requerimientos hacia su próxima representación en el ciclo de vida.
  3. Composición.
  4. Evolución.
  5. Escalabilidad.
  6. Soporte para decisión y negociación de requerimientos.
  7. Mapeo de requerimientos hacia elementos posteriores (decisiones, estructuras, procedimientos).
  8. Homogeneidad.
  9. Verificación y validación.

Cuadro 5.2 Características de los enfoques para ingeniería de requerimientos con separación de intereses. (Chitchyan, y otros, 2005).

Con base en los enfoques actuales de ingeniería de requerimientos orientados a aspectos, se han identificado tendencias que marcan un proceso emergente para los trabajos futuros. En la Figura 5.1 se muestra este proceso gráficamente. Se inicia con una etapa de extracción, cuyo objetivo es obtener una lista preliminar de requerimientos a partir de las necesidades de los usuarios. Posteriormente en la etapa de identificación se define el campo de acción de los requerimientos; en la etapa de representación se modelan utilizando algún enfoque, por ejemplo, puede ser alguno de los que se encuentran listados en el Cuadro 5.1 y en el Cuadro 5.2 . La fase de refinamiento en conjunto con las anteriores se repite hasta tener suficientes intereses representados.

Posteriormente comienza la etapa de composición, la cual revelará los posibles conflictos, que deben ser resueltos mediante la negociación, es decir, el intercambio y la modificación de los requisitos y sus alcances. Esta etapa continúa iterativamente con las anteriores hasta que se logra un conjunto estable de requerimientos. Es entonces cuando inicia el mapeo, cuyo objetivo es enlazar los requerimientos con los productos de las etapas de arquitectura y diseño.

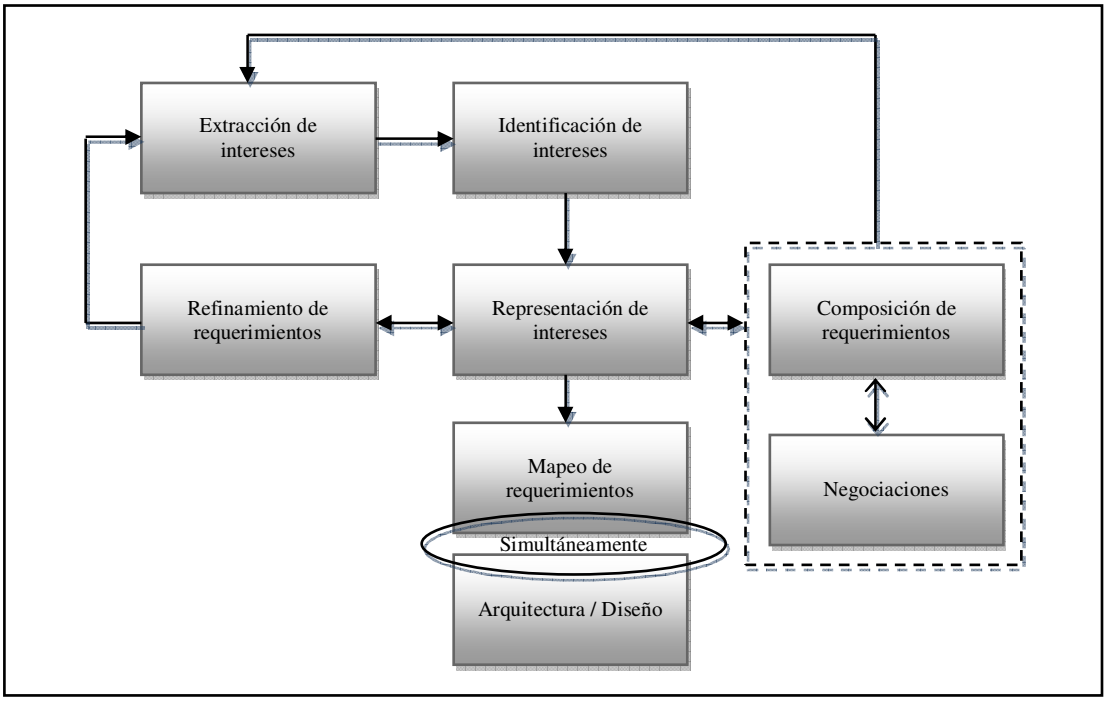


Figura 5.1. Proceso emergente de ingeniería de requerimientos orientada a aspectos. (CHITCHYAN, Ruzanna et al., 2005).

5.4. Aspectos en la arquitectura

La arquitectura es una representación de alto nivel que facilita la comunicación entre las personas involucradas en el desarrollo de un sistema mostrando una estructura del sistema que incluye componentes de software, las propiedades externamente visibles de esos componentes y las relaciones entre ellos. (CHITCHYAN, Ruzanna et al., 2005).

La arquitectura captura las decisiones de diseño más tempranas e incluye componentes de alto nivel que tienen impacto en las posteriores actividades de análisis, diseño e implementación. Los métodos tradicionales de arquitectura de software no distinguen explícitamente los intereses arquitecturales convencionales de los transversales. Esto trae como consecuencia el riesgo de pasar por alto ciertos aspectos potenciales que pueden

degradar la calidad del sistema conforme se avanza en el proceso de desarrollo. De ahí la necesidad de mecanismos explícitos para identificar, especificar y evaluar aspectos al nivel de diseño de la arquitectura.

El diseño de arquitectura orientado a aspectos también ha sido un campo de trabajo fértil en los últimos años; hasta ahora las principales contribuciones realizadas en esta área han sido: mostrar la factibilidad de utilizar lenguajes de descripción arquitectural orientados a aspectos (AO-ADLs, como DAOP-ADL(PINTO, M. et al., 2003), por ejemplo), proponer medios para lograr la trazabilidad de los aspectos de la arquitectura hacia la etapa de implementación, evaluar arquitecturas de software con ayuda de la separación multidimensional de intereses, y destacar la importancia de utilizar varias vistas diferentes para enfrentar el problema del diseño de la arquitectura de un sistema.

De la misma manera como sucede en la ingeniería de requisitos, en la etapa de arquitectura orientada a aspectos existen diferentes metodologías aún sin consenso unificado; sin embargo, ya se vislumbran marcadas tendencias de acuerdo a los trabajos realizados hasta ahora. El proceso de trabajo emergente para el diseño de arquitectura orientada a aspectos se muestra en la Figura 5.2. El proceso inicia cuando se tienen ya la especificación, los aspectos y las negociaciones provenientes de la etapa de requerimientos. Se van refinando estos requerimientos mientras que se identifican aspectos de arquitectura adicionales mediante el análisis de dominio. Después, se utiliza algún enfoque para modelar la arquitectura, la cual posteriormente es evaluada para realizarle adecuaciones y mejoras. Los procesos de modelado, evaluación y composición así como el de la refactorización son iterativos. Una vez que se tiene una arquitectura estable, se realiza la relación de trazabilidad correspondiente con los productos de las fases de diseño e implementación.

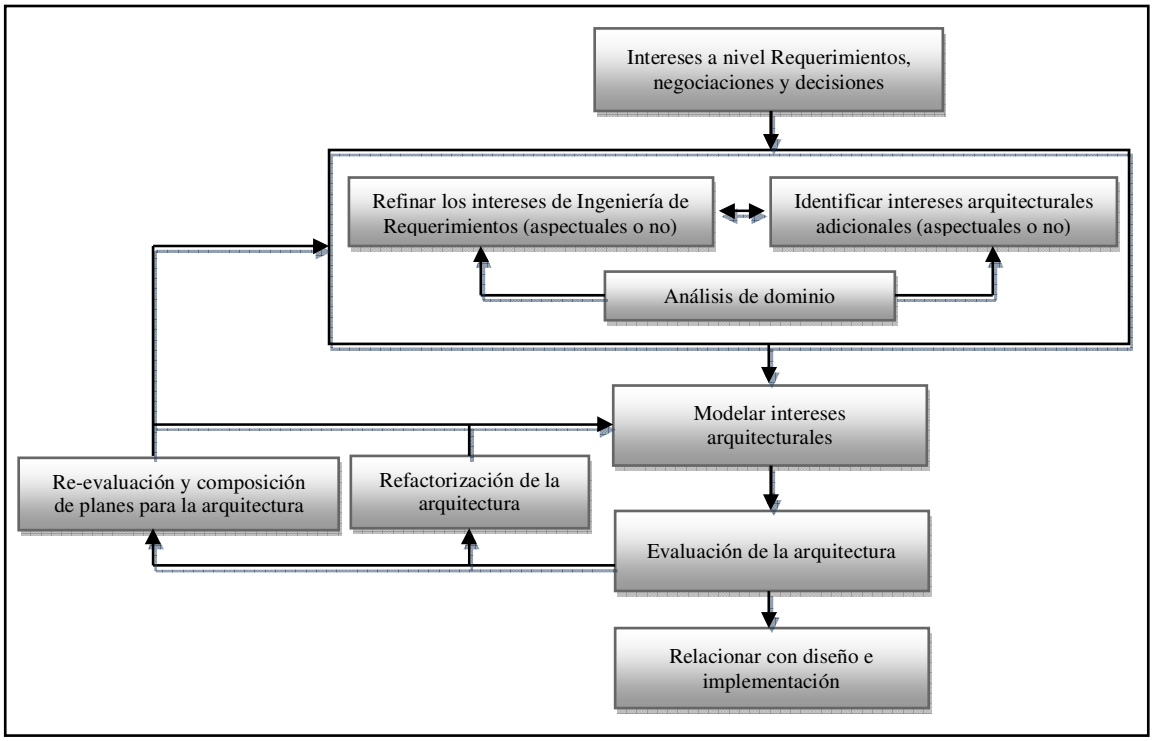


Figura 5.2. Proceso emergente de diseño de arquitectura orientada a aspectos. (CHITCHYAN, Ruzanna et al., 2005).

5.5. Aspectos en el diseño

El diseño es la actividad técnica que sigue a la determinación de la arquitectura del sistema. Su meta es preparar el proyecto para que sea implementado. Es el diseño quien proporciona los detalles de la solución considerando los requerimientos y las decisiones tomadas en la fase de arquitectura.

En proyectos de software orientados a aspectos como en (CLEMENTE, Pedro et al., 2004), se ha concluido como resultado de la experiencia, que la orientación a aspectos debe tomarse seriamente en cuenta en todas las etapas de desarrollo, pero de manera particular debe estar presente en el diseño.

Se ha observado en estos proyectos que los programadores tradicionales tienen dificultades implementado programas orientados a aspectos hasta que adquieren cierta destreza. Estos problemas tan marcados se deben en gran parte a que el diseño del sistema se realiza sin considerar los aspectos, de tal manera que el programador debe rediseñar partes del sistema para introducir la separación de intereses dispersos. Esto se vuelve un problema aún mayor cuando en algunas organizaciones existen políticas que eximen al programador de las tareas de diseño.

Las tareas de re-diseño forzado para introducir aspectos en los programas resultan en responsabilidades ambiguas entre el diseñador y el programador, especialmente cuando el software final presenta problemas. Realizar diseño sin orientación a aspectos para un programa orientado a aspectos conduce a serios inconvenientes como la falta de coherencia con el diseño original; además, el mantenimiento y la evolución del sistema se tornan más difíciles, ya que el programador desconoce si los cambios que él introduce afectarán a otras partes del sistema.

Estos problemas no deben hacer suponer que la POA deba evitarse, sino más bien dan paso a considerar que debe ser utilizada en conjunto con adecuadas técnicas de diseño orientado a aspectos, ya que la carencia de estas últimas se identifica como la principal causa de inconsistencias en los programas. El diseño orientado a aspectos tiene por objetivo especificar la estructura y comportamiento del sistema de software. Un enfoque de diseño orientado a aspectos debe proporcionar las construcciones necesarias para la modularización de intereses, sin importar si son transversales o no; también debe de soportar la composición de intereses, considerando opciones de solución para posibles conflictos.

Un enfoque de diseño orientado a aspectos generalmente incluye un proceso y un lenguaje. El proceso toma los requerimientos como entrada y produce un modelo de diseño adecuado a la arquitectura del sistema. El modelo resultante incluye intereses y sus relaciones, los cuales pueden implementarse en una plataforma orientada a aspectos o no. El lenguaje incluye construcciones capaces de describir los elementos a ser representados en el diseño y sus posibles relaciones; lo cual permite especificar la composición de los intereses.

La experiencia y el estudio de las técnicas existentes de diseño orientadas a aspectos han enriquecido el área de la Ingeniería de Software. Se pueden identificar las siguientes como las contribuciones más importantes del diseño orientado a aspectos:

- Proporcionar a los diseñadores diversas maneras para modelar sistemas con un enfoque orientado a aspectos.
- Permitir razonar en la etapa de diseño acerca de los intereses de manera separada, independientemente de si se trata de un interés transversal o no.
- Capturar modularmente especificaciones de diseño para los intereses.
- Proveer formas de especificación para la cooperación y composición de módulos.
- Proporcionar maneras para resolver los conflictos que surgen entre los intereses.

Con base en la tendencia de las técnicas y prácticas actuales, en la Figura 5.3 se muestra el horizonte del proceso emergente de diseño orientado a aspectos. El proceso consta de tres niveles que permiten ver el diseño desde perspectivas generales o específicas. En el nivel más alto, los diseños son abstractos y sin detalle. Por otra parte, en el nivel medio, los diseños muestran más detalles, pero siempre lo hacen independientemente de la plataforma utilizada. Finalmente, en el nivel más bajo, los detalles son dependientes de una plataforma específica.

Manteniendo el respectivo grado de abstracción, se deben modelar los componentes transversales, los no transversales y las relaciones de composición en cada uno de los tres niveles. Como el proceso de composición está siempre presente en cada nivel, es posible tener una visión más clara del impacto de las interacciones de los elementos sobre el diseño global en cada momento del proceso.

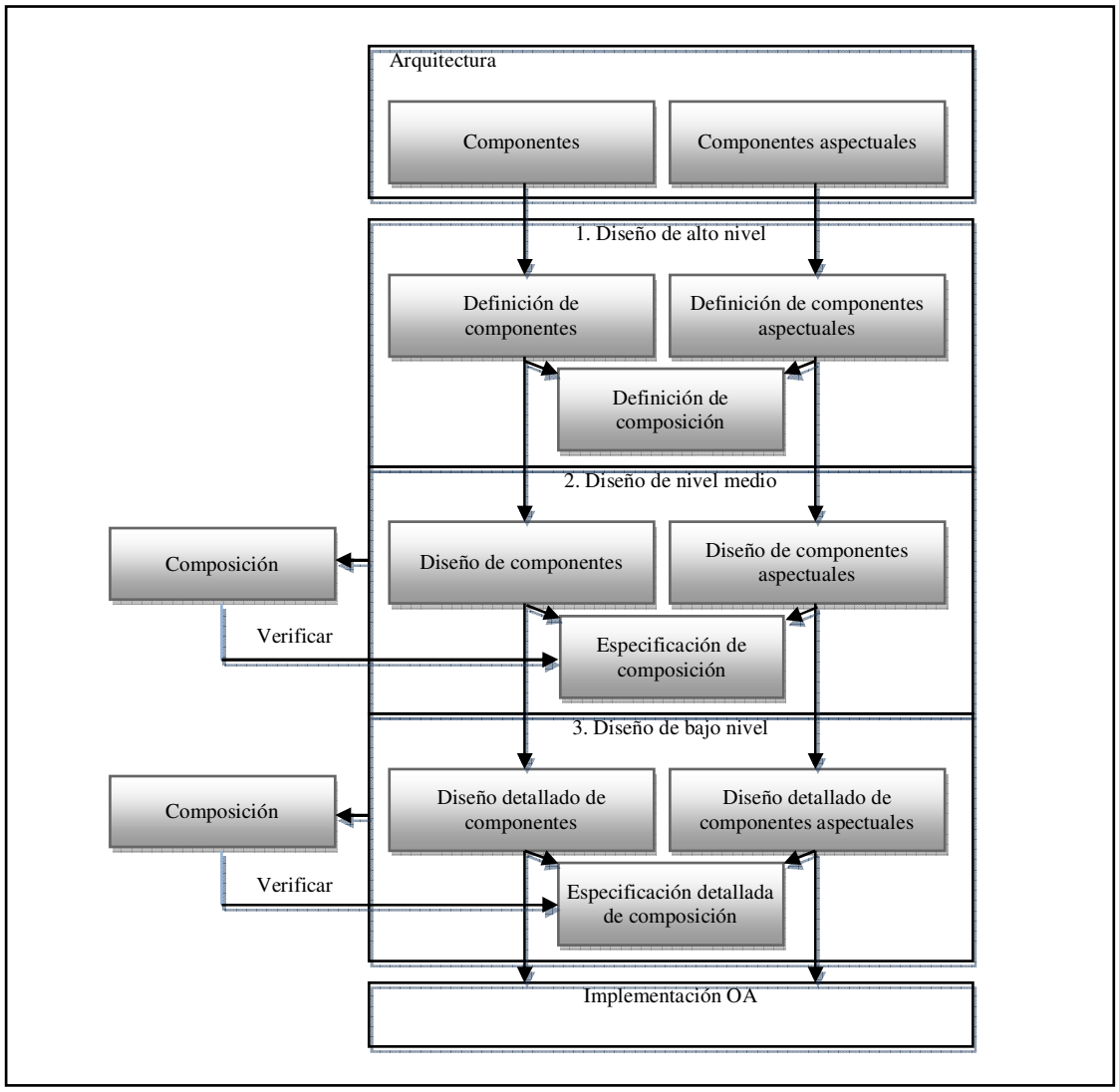


Figura 5.3. Proceso emergente de diseño orientado a aspectos. (CHITCHYAN, Ruzanna et al., 2005).

5.6. Conclusiones

El estudio de los aspectos tempranos abarca las etapas de análisis de dominio, ingeniería de requerimientos, arquitectura y diseño. Los diferentes trabajos realizados en este ámbito buscan una adecuada separación de intereses con características deseables como posibilidad de generar un modelo compuesto, trazabilidad hacia las diferentes actividades de desarrollo, y facilidad para la evolución y escalabilidad de las especificaciones.

El estudio de los aspectos tempranos parece abrir paso a un ciclo de vida más homogéneo donde una implementación orientada a aspectos no se encuentra desligada de su previo análisis y diseño. Esto proporciona medios para superar las inconsistencias introducidas por la mezcla de actividades orientadas a aspectos con otras que no lo son.

Parte II.
El trabajo de tesis

Capítulo 6. Presentación

6.1. Introducción

La idea orientadora del presente trabajo surgió al identificar un área de estudio poco abordada en la Ingeniería de requisitos que conjuga la especificación formal de requerimientos con lenguaje Z, la separación modular de intereses con soporte aspectual y la reutilización de especificaciones. Z-SIIMD (Z con Separación de Intereses con Influencia Multidimensional) es una propuesta que encajaba en el área de estudio identificada., proponiendo un modelo de trabajo elementos: requerimientos, intereses, reglas de composición, y los procesos de composición y reutilización. Posteriormente se vio la necesidad de contar con un software que implementara las características más relevantes de Z-SIIMD, y fue así como se desarrolló SPZS (Software Prototipo para Z-SIIMD).

En Z-SIIMD se trabaja con una separación de intereses con un modelo con influencia multidimensional desde el inicio del desarrollo; se destaca el hecho de que no es necesario desde un principio separar los intereses en dos modelos: base y aspectual, como sucede en el modelo bidimensional. De esta manera, la descomposición principal, o dominante que se elija no afecta la forma de trabajar con los intereses, y los elementos se pueden modularizar de una manera más limpia.

Al utilizar intereses, se está eligiendo una descomposición inicial influenciada por las necesidades de ese proyecto en particular, que sin duda contribuye al mejor entendimiento de las situaciones que están siendo especificadas, pero que podría restringir las oportunidades de reutilización. Sin embargo, en Z-SIIMD se propone que los requerimientos individuales o el mismo interés completo sean elementos reutilizables que se puedan incorporar en otros proyectos, superando esa limitante. Además, se puede tomar ventaja del uso de intereses para especificar, organizar y revisar más fácilmente las interacciones de los requerimientos que tienen influencia sobre intereses completos, lo cual se consigue administrando una serie de reglas que se encuentran aisladas del resto de la especificación. La organización de los requerimientos en intereses es un área de oportunidad que permitiría en un futuro trabajar con múltiples descomposiciones simultáneamente en un proyecto.

De manera general, al trabajar con un proyecto en Z-SIIMD (Figura 6.1) primero, todos los requerimientos se especifican como unidades aisladas con entradas, salidas y estados; luego, opcionalmente se agrupan en intereses comunes de acuerdo al grado de cercanía conceptual que comparten. Posteriormente, se escriben reglas para determinar la manera en la que los requerimientos se afectan entre sí. Finalmente, mediante las reglas, se combinan

los requerimientos y se genera un modelo integrado; una vez que se obtiene este modelo, es posible agregar más requerimientos, intereses o reglas de composición al proyecto y volver a generar el modelo compuesto para reflejar las modificaciones realizadas.

Los elementos principales de Z-SIIMD son: requerimientos, intereses, reglas de composición, los procesos de integración y reutilización, y el software SPZS (Software Prototipo para Z-SIIMD). En los siguientes apartados se presentará brevemente cada uno de ellos.

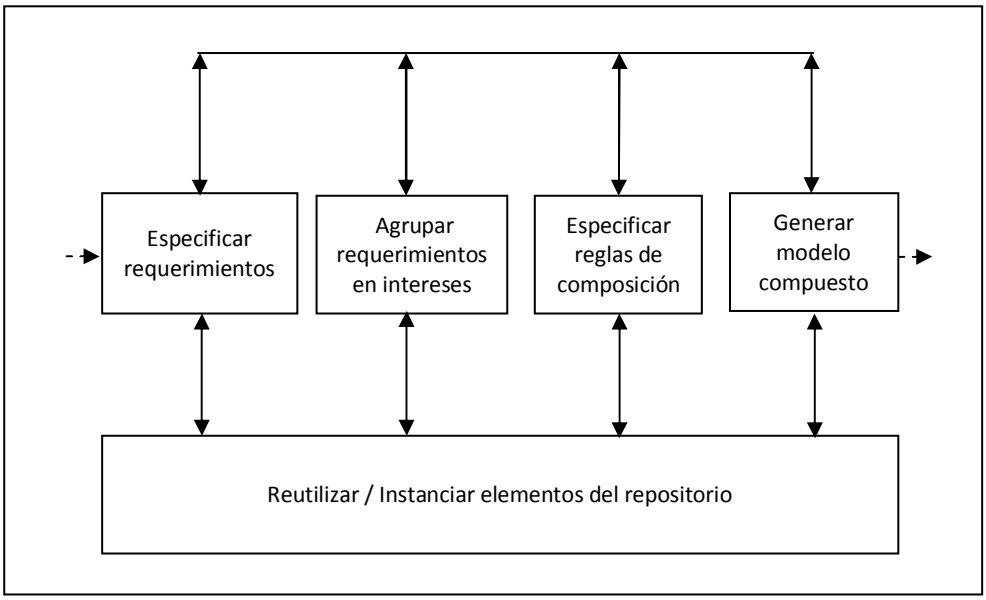


Figura 6.1 Modelo de trabajo de Z-SIIMD

6.2. Requerimientos e intereses

Un requerimiento Z-SIIMD es una característica particular que el software debe cumplir; tiene datos asociados y una especificación formal en Z, que conforman su descripción. Un interés Z-SIIMD es un grupo de requerimientos que por su naturaleza común, resultan idóneos para ser reunidos en un solo módulo y darles seguimiento durante el resto de las actividades del ciclo de vida. Los intereses Z-SIIMD tienen como objetivo organizar conceptualmente los requerimientos de un proyecto. Esto puede ser beneficioso cuando un requerimiento afecta a varios al mismo tiempo; en este caso, se podría escribir una sola regla, y no una para cada requerimiento afectado. Las reglas de composición Z-SIIMD pueden trabajar con intereses y requerimientos, o solo con requerimientos.

Para describir los requerimientos se propone un esquema de especificación en el que se emplea lenguaje Z apoyado con lenguaje natural. Si bien es cierto que los lenguajes

formales permiten especificaciones precisas, concretas y simbólicas, también es bien sabido que el uso de lenguaje natural se considera una útil documentación y hace que tanto el desarrollador como el usuario entiendan el proceso de una mejor manera en cualquier momento.

Adicional a su especificación formal, los requerimientos Z-SIIMD tienen otra información asociada que ayuda a describirlos, organizarlos y ubicarlos mejor. Es así como un requerimiento posee también, por ejemplo, descripción en lenguaje natural, categorías, palabras clave, registros de trazabilidad y de evolución. Los intereses Z-SIIMD también contienen esta información (consultar el modelo completo de requerimientos e intereses en el apartado 7.2).

Una vez que los requerimientos han sido capturados e identificados de manera separada, fuera o dentro de intereses, se deben especificar las influencias transversales. Se emplea el término transversal porque aunque la especificación del requerimiento se encuentra aislada, atraviesa a uno o más requerimientos del proyecto a la vez. Son las reglas de composición Z-SIIMD las encargadas de describir el impacto transversal de los requerimientos entre sí.

6.3. Reglas de composición

Una regla de composición especifica la manera en la que los requerimientos se combinan con otros. Cuando se conocen los requerimientos que atraviesan a otros así como los que resultan atravesados, ya se pueden escribir las reglas de composición, las cuales van a entrelazar los elementos necesarios para producir un modelo compuesto con las adecuadas especificaciones combinadas. El proceso de tomar dos requerimientos y combinarlos adecuadamente de acuerdo a directrices pre-determinadas se denomina composición. Las sentencias que contienen las instrucciones para realizar este entrelazado se llaman reglas de composición.

Las reglas de composición Z-SIIMD contienen dos operandos: el primero es el requerimiento que atraviesa al (los) requerimiento(s) denotado(s) por el segundo operando. El primer operando es siempre un solo requerimiento, mientras que el segundo puede ser un requerimiento individual, o un conjunto de ellos.

Además de indicar los requerimientos involucrados en la composición, las reglas proporcionan los detalles para que este proceso pueda llevarse a cabo. Por ejemplo, contienen parámetros que configuran la relación lógica entre los requerimientos involucrados, y la ubicación en el modelo compuesto.

Las reglas además permiten incluir declaraciones de nuevas variables y condiciones en el modelo compuesto, sustituir algunas variables involucradas por valores específicos y cambiar la naturaleza de las variables (por ejemplo, una variable de entrada en un

requerimiento puede pasar a ser una variable de trabajo en el requerimiento compuesto resultante). Las reglas de composición con sus elementos se describen en el capítulo 7.

6.4. El proceso de composición

El proceso de composición Z-SIIMD toma como entradas los intereses, requerimientos y las reglas que se incluyen en una especificación para producir un modelo compuesto, también llamado completo o integrado, el cual contiene los efectos de las reglas de composición sobre los requerimientos. En el modelo compuesto se incluyen todos los requerimientos que fueron especificados, y aquellos que fueron generados a partir de las reglas.

Las reglas son las guías para el proceso de composición, pues son las que determinan los diferentes cursos de acción del algoritmo. Cada cambio en las reglas tiene repercusiones en el modelo compuesto obtenido al final. Es por esto que se visualiza al proceso de composición y a las reglas como un binomio fuertemente acoplado que trabaja siempre en conjunto. El proceso de composición se aborda en el Capítulo 8; y en el Capítulo 11 se muestra su aplicación práctica en un caso de uso.

6.5. El proceso de reutilización

Existen comportamientos que por su frecuente aparición en diversos sistemas, pueden ser abstraídos, almacenados y posteriormente utilizados de nuevo. En este caso, el esfuerzo para especificar estas funcionalidades se enfoca en obtener la primera vez elementos correctamente aislados y capturados para después simplemente reutilizarlos y adaptarlos.

Cualquiera de los elementos Z-SIIMD incluidos en un proyecto se pueden reutilizar: requerimientos, intereses o reglas de composición. Por ejemplo, un requerimiento debidamente especificado puede ser almacenado en el repositorio de elementos categorizados, para posteriormente ser copiado íntegramente en la especificación de otro proyecto, donde se puede adecuar para su uso en ese escenario particular. El repositorio representa una biblioteca común para todos los proyectos, donde los elementos se encuentran organizados por su naturaleza, categoría y palabras clave que permitan ubicarlo fácilmente.

El proceso de reutilización funciona en dos fases: en la primera, denominada clasificación, el desarrollador toma elementos del proyecto actual y los agrega al repositorio, debidamente etiquetados. En la segunda fase, denominada instanciación, un proyecto puede incorporar en sí mismo copias de elementos que ya existen en el repositorio. En esta segunda fase, los elementos deben ser seleccionados y adecuados por el desarrollador para que puedan ser incorporados sin conflictos al proyecto actual.

La biblioteca de elementos y el proceso de reutilización se requieren mutuamente para ser útiles. El proceso permite la clasificación, búsqueda y gestión de los elementos del repositorio, que actúa como un contenedor al que se puede acceder para la reutilización. El modelo de reutilización se explica en el Capítulo 9, y en el Capítulo 12 se muestra un ejemplo de su aplicación a un caso de uso.

6.6. El software SPZS

Las principales características de Z-SIIMD fueron implementadas en un software prototipo denominado SPZS, el cual fue desarrollado exclusivamente para dar soporte a esta tesis. SPZS posee una interfaz gráfica que permite al especificador trabajar con proyectos donde se introducen los requerimientos, intereses y reglas de composición. SPZS automatiza el proceso de composición, genera un modelo final integrado, y proporciona repositorios que son gestionados por el proceso de reutilización. Este software también implementa los procesos de clasificación e instanciación de elementos Z-SIIMD.

El software se inició como un prototipo incremental que aún se encuentra en evolución. En su diseño se realizaron varias consideraciones para delimitar su funcionalidad y concentrar el esfuerzo de desarrollo en la obtención de resultados estratégicos. Estas consideraciones así como las principales características de SPZS se presentan en el Capítulo 10.

6.7. Organización

Esta parte del trabajo se organiza de la siguiente manera: El Capítulo 7 presenta la descripción detallada de los elementos del modelo de trabajo de Z-SIIMD y la sintaxis de la extensión propuesta de Lenguaje Z. El Capítulo 8 explica el algoritmo de composición utilizado para generar un modelo final completo. En el Capítulo 9 se explican las características de los procesos de clasificación e instanciación. El Capítulo 10 contiene la descripción de las características de SPZS, el software prototipo desarrollado para esta tesis. En el Capítulo 11 se ejemplifica el proceso de composición de especificaciones con un caso de uso para un agrupamiento de conexiones a bases de datos. Finalmente, en el Capítulo 12 se presenta un caso de uso para un retiro bancario con el objetivo de mostrar la aplicación de los procesos de clasificación e instanciación.

Capítulo 7.
Elementos del modelo de trabajo

7.1. Introducción

El modelo de trabajo general de Z-SIIMD se muestra gráficamente en la Figura 7.1. Un proyecto de especificación involucra requerimientos, intereses y reglas de composición. La descripción de los requerimientos se realiza de manera modular, y se organiza opcionalmente en conceptos afines dentro de ese proyecto que se denominan intereses. Los intereses pueden agrupar uno o varios requerimientos al mismo tiempo. Separadamente se escriben las reglas de composición, las cuales determinan los detalles acerca de la forma en la que los requerimientos influyen sobre otros.

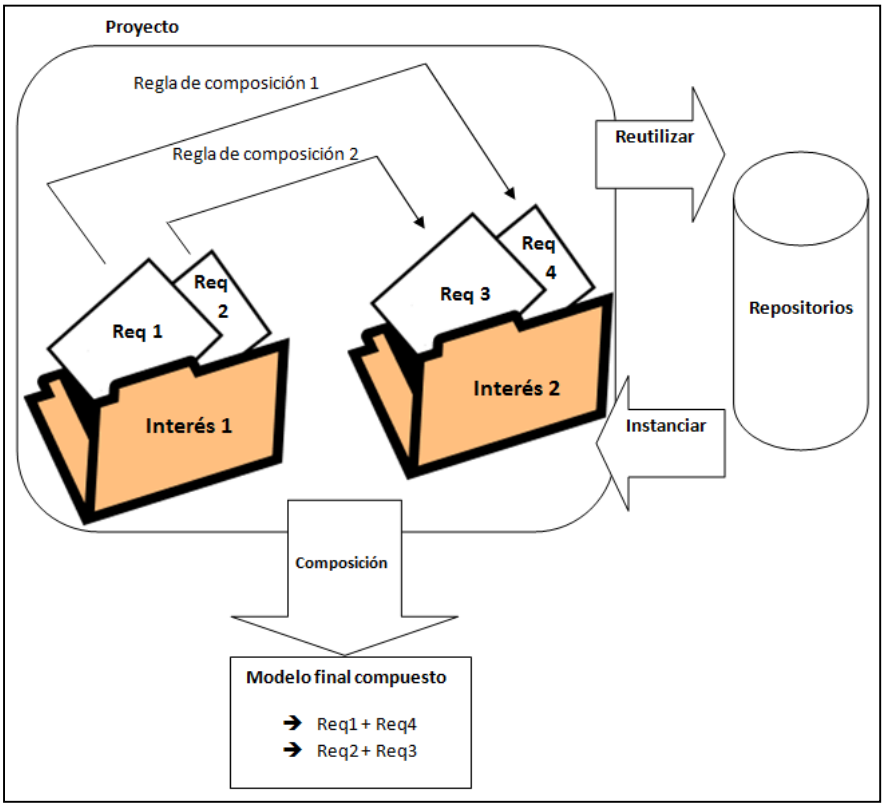


Figura 7.1 Esquema de separación de intereses del modelo de trabajo de Z-SIIMD.

En el modelo de trabajo existen dos procesos relevantes: la composición y la reutilización. El objetivo del primero es generar un modelo completo a partir de los elementos definidos aisladamente en el proyecto. La composición es guiada por las reglas. El objetivo del segundo es permitir el acceso al repositorio. El acceso se lleva a cabo en dos momentos: 1) Cuando se identifica que un elemento del proyecto actual puede ser reutilizado

posteriormente y se almacena en el repositorio; este momento se denomina clasificación. 2) Cuando se localiza un elemento en el repositorio que es útil para el proyecto actual, se incorpora en él y se adecúa para su uso en ese contexto; este segundo momento se denomina instanciación.

En este capítulo se detallan los respectivos modelos utilizados para intereses, requerimientos y reglas de composición (en el apartado 7.2), se enfatizan las posibilidades brindadas por las reglas (en el apartado 7.3), y se muestra la sintaxis de la extensión de lenguaje Z de Z-SIIMD (en el apartado 7.4).

7.2. Modelo de los elementos

Las representaciones de intereses, requerimientos y reglas tienen características similares, las cuales se han agrupado en el "modelo base" que se muestra en la Figura 7.2. Éste modelo consiste en un identificador único, una descripción en lenguaje natural, datos de clasificación, de evolución y trazabilidad. El identificador es la identidad de cada elemento que sirve para hacer referencia unívoca a él cuando sea necesario, por eso debe ser único dentro de un mismo proyecto.

Los datos de clasificación permiten relacionar cada elemento con una o varias categorías, y agregar las palabras clave que estén relacionadas con su funcionalidad. El propósito de las categorías y palabras clave es ayudar a ubicar elementos en el repositorio cuando se realizan búsquedas de elementos. Los datos de clasificación los determina libremente el desarrollador en base a la descripción de las necesidades particulares de la aplicación en la cual trabaja.

Los datos de evolución sirven para llevar una bitácora de los cambios realizados a cada elemento en el proyecto y en el repositorio.

Los datos de trazabilidad tienen como objetivo servir como enlace entre la especificación y el resto de las etapas del ciclo de desarrollo. De esta manera, por ejemplo, un requerimiento puede convertirse en una característica funcional o no funcional del sistema, o en una decisión de diseño.

- Identificador (Nombre único que lo identifica del resto de los elementos)
- Descripción en lenguaje natural
- Datos de clasificación

Categorías
…

Palabras clave
…

- Evolución (Bitácora de cambios)

Numero de Versión (Identificador de versión)	Tipo (Agregado/ Borrado/ Modificado)	Fecha	Hora	Elemento afectado	Valor anterior	Valor nuevo
…	…	…	…	…	…	…

- Trazabilidad (durante las actividades del ciclo de vida)

Fecha y Hora	Actividad*	Mapeo**	Comentario
…	…	…	…

*Actividad → Relacionada con: Especificación / Arquitectura / Diseño / Implementación /Prueba / Evolución.

**Mapeo → Característica del sistema / Decisión / Aspecto.

Figura 7.2 Modelo base de los elementos

Además de las características comunes mencionadas anteriormente, los requerimientos incorporan una representación formal que consiste en dos secciones: declaraciones y predicados (o invariantes). Esta representación corresponde al concepto de esquema de Lenguaje Z explicado en el apartado 14.17. Adicionalmente a la representación formal, se proporciona un campo para describir en lenguaje natural el propósito de cada una de las secciones. El resumen de los elementos que completan el modelo para un requerimiento se muestra en la Figura 7.3.

- Representación formal
  - Declaraciones

Declaración	Descripción en lenguaje natural
…	…

  - Invariantes

Invariante	Descripción en lenguaje natural
…	…

Figura 7.3 Elementos que completan el modelo para un requerimiento.

Además de las características del modelo base, un interés incorpora una lista de requerimientos asociados. Dentro de un interés, cada requerimiento tiene la estructura previamente mostrada en el modelo base, y adicionalmente, contiene un texto para incluir un comentario, el cual resulta útil para documentar situaciones particulares al contexto del interés en el que se encuentra. Los elementos que completan el modelo para un interés se muestran en la Figura 7.4.

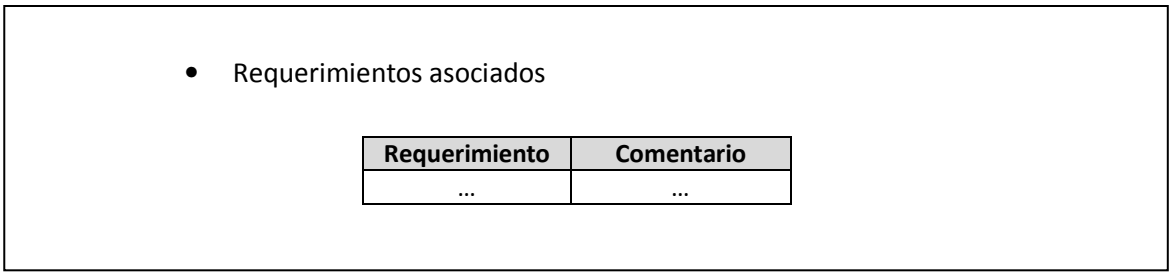


Figura 7.4 Elementos que completan el modelo para un interés.

El elemento unificador de los requerimientos, intereses y reglas que conforman una especificación es el proyecto. Un proyecto contiene en su totalidad la información aislada y ordenada que es necesaria para generar un modelo compuesto y completo. Puede concebirse como un grupo de elementos reunidos en un solo contenedor. El trabajo con cualquier elemento de una especificación en particular, ya sea requerimiento, interés o regla, en realidad se lleva a cabo dentro del marco de un proyecto.

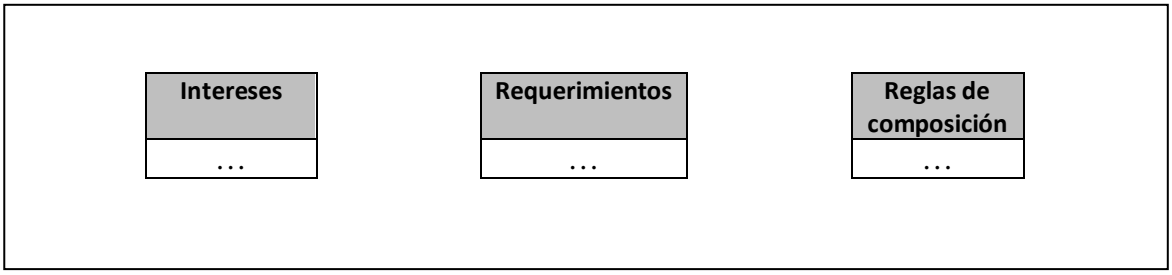


Figura 7.5 Elementos de un proyecto de especificación.

7.3. Modelo para una regla de composición

La forma general que toman las reglas de composición dentro de Z-SIIMD es la siguiente:

Interés$_m$.Requerimiento$_n$ [ORIG | COMP] **@CC** Interés$_r$.Requerimiento$_s$ [ORIG | COMP]
@WHERE DescriptorLugar
@CONN OpLog
[**@NEWNAME** "nombreNuevoRequerimiento"]
[**@ADDVAR** ("varA$_1$: declaracionA$_1$"; … ;
"varA$_n$: declaracionA$_n$";)]
[**@COND** ("CondiciónDeComposición$_1$"; … ;
"CondiciónDeComposición$_n$");]
[**@CHG** (varCH$_1$ = TipoVar$_1$; … ; varCH$_n$ = TipoVar$_n$;)]
[**@REP** ("varR$_1$ = valorR$_1$"; … ; "varR$_n$ = valorR$_n$";)]

La descripción de cada uno de los términos involucrados se encuentra en el Cuadro 7.1.

Elemento	Descripción o Significado
Interés	Nombre del interés
Requerimiento	Nombre del requerimiento
ORIG COMP	Cláusulas que permiten referirse a la versión original (ORIG) o a la última versión compuesta (COMP) – más reciente al momento de aplicar la regla – de los requerimientos involucrados en la regla de composición.
@CC	Cláusula que significa "**C**ross**C**uts" (atraviesa a), se utiliza justo entre los dos intereses involucrados (el primero atraviesa al segundo).
@WHERE	Cláusula que permite indicar un valor para DescriptorLugar.
DescriptorLugar	Descriptor que indica el lugar en donde se debe llevar a cabo la composición.
@CONN	Cláusula que permite introducir el operador lógico que relaciona a los requerimientos de la regla.
OpLog	Operador lógico que indica la manera de relacionar ambos requerimientos (Y / O / Y NO / O NO).
@NEWNAME	Cláusula que permite especificar un nombre para el nuevo requerimiento generado.
nombreNuevoRequerimiento	Nombre que se va a asignar al nuevo requerimiento generado.
@ADDVAR	Cláusula que permite agregar nuevas variables al requerimiento compuesto resultante.
varA$_n$	Nombre de la variable que se va a agregar al requerimiento compuesto.
declaracionA$_n$	Declaración del tipo de la variable que se va a agregar al requerimiento compuesto.
@COND	Cláusula que permite agregar una condición especial para que un requerimiento "atraviese" al otro.
CondiciónDeComposición	Es una condición especificada que se agrega en el momento de la composición y puede involucrar elementos de ambos requerimientos y operadores lógicos correctamente ubicados.
@CHG	Cláusula opcional que permite cambiar la naturaleza de las variables.
varCH$_n$	Nombre de una variable a la que se desea cambiar su naturaleza.
TipoVar$_n$	Constante que indica la naturaleza de las variables (Entrada / Salida / Trabajo).
@REP	Cláusula opcional que permite reemplazar valores de las variables esperadas de los requerimientos.
varR$_n$	Nombre de una variable a la que se desea asignar un valor al realizarse la composición.
valorR$_n$	Valor que se asigna a una variable al realizarse la composición.

Cuadro 7.1 Elementos de una regla de composición.

7.3.1. Opciones de las reglas de composición

Varios requerimientos afectados con la misma regla

Es posible que un requerimiento atraviese a varios requerimientos. Esto se logra incluyendo el operador asterisco (*) en el segundo operando de la regla. El uso del asterisco es aceptado solo si aparece en el segundo operando, acompañado previamente del nombre de un interés con más de un requerimiento asociado.

Entonces, siguiendo la sintaxis de las reglas de composición, el operador asterisco (*) puede incorporarse solamente en lugar de $\texttt{Interes}_r\texttt{.Requerimiento}_s$ (segundo operando) de la siguiente manera: $\texttt{Interes}_r\texttt{.*}$ así, es posible expresar con una sola regla el hecho de que un requerimiento atraviese a todos los requerimientos (*) contenidos en $\texttt{Interes}_r$. La siguiente regla de composición ejemplifica este caso. Obsérvese que el requerimiento `AgregarABitacora` contenido dentro del interés `Bitacora`, atravesará a todos los requerimientos que se encuentran en el interés llamado `Retiro`.

```
@COMPOSITIONRULE
@ID "BitacoraAlRetiro";
@RULE
Bitacora.AgregarABitacora ORIG      @CC   Retiro.* ORIG
                                    @WHERE AFT
                                    @CONN AND
```

Versiones originales y compuestas de los requerimientos

Una regla de composición permite que los requerimientos sean atravesados por un solo requerimiento a la vez. Sin embargo, es posible mediante la acción acumulada de varias reglas separadas que un requerimiento sea afectado por varios requerimientos distintos; entonces se debe decidir cuál es la versión del requerimiento que se debe considerar al momento de generar un requerimiento compuesto.

En una regla de composición se puede especificar la versión del requerimiento al que se hace referencia. Esto se hace con las constantes `ORIG` y `COMP`. `ORIG` denota la versión original del requerimiento, mientras que `COMP` implica la versión compuesta más reciente de ese requerimiento. Si cualquiera de las dos constantes se omite, se asume la versión compuesta más reciente (`COMP`).

`ORIG` y `COMP` funcionan de la siguiente manera: si un requerimiento `R` es afectado por las reglas de composición `RC1` y `RC2` consecutivamente, el uso de `COMP` en `RC2` permitiría que `RC2` actuara sobre una versión de `R` que contiene los efectos de `RC1`. El uso de `ORIG` en `RC2` permitiría que `RC2` actuara directamente sobre la versión original de `R` sin ser

afectada por RC1. A continuación se muestra un ejemplo de esta sintaxis con una regla de composición. Esta regla indica que un requerimiento llamado IdentificacionDeUsuarios ubicado dentro del interés nombrado Seguridad atraversará al requerimiento Captura, ubicado dentro del interés con nombre Actividades. La palabra ORIG después del nombre de los requerimientos indica que ambos participarán en el proceso de composición con sus versiones originales.

```
@COMPOSITIONRULE
@ID "AgregarSeguridadACaptura";
@RULE
Seguridad.IdentificacionDeUsuarios ORIG
                                   @CC   Actividades.Captura ORIG
                                   @WHERE BEF
                                   @CONN AND
```

Al trabajar con las versiones compuestas de los requerimientos se puede pensar en un sistema de prioridad implícito con efectos acumulativos. Para explicarlo mejor, considérese el siguiente ejemplo: al finalizar la ejecución de un programa escrito en un lenguaje de programación, se alcanza un estado con el efecto acumulado que resulta de la ejecución secuencial de cada una de sus instrucciones; la sentencia que se encuentra ubicada primero en el programa es la primera que se ejecuta y el efecto que causa la ejecución de ésta, provoca un estado intermedio que la siguiente secuencia recibe y puede modificar.

De la misma manera sucede con la composición usando versiones compuestas de requerimientos: cada regla describe aisladamente la manera en que un requerimiento atraviesa a otro. Al final del proceso de composición, se obtiene un modelo que contiene los requerimientos con los efectos acumulados de cada regla del modelo. La regla que aparece primero en la especificación es la primera que se ejecuta y el efecto que ésta provoca en el modelo completo se mantiene para la siguiente regla.

Ubicación de los elementos al realizar la composición

Cuando se van a combinar dos requerimientos en uno nuevo compuesto, es necesario especificar la ubicación en la que aparecerán las declaraciones y condiciones del primer requerimiento respecto al segundo. Esta información se proporciona mediante la sentencia @WHERE y el valor DescriptorLugar.

La constante DescriptorLugar utiliza uno de los términos constantes del Cuadro 7.2.

Constante	Significado
BEF	El contenido de $\texttt{Interés}_m\texttt{.Requerimiento}_n$ será insertado y expandido antes del contenido de $\texttt{Interés}_r\texttt{.Requerimiento}_s$
AFT	El contenido de $\texttt{Interés}_m\texttt{.Requerimiento}_n$ será insertado y expandido después del contenido de $\texttt{Interés}_r\texttt{.Requerimiento}_s$
REP	El contenido de $\texttt{Interés}_m\texttt{.Requerimiento}_n$ será insertado y expandido reemplazando al conenido de $\texttt{Interés}_r\texttt{.Requerimiento}_s$
NEWBEF	El nombre del requerimiento $\texttt{Interés}_m\texttt{.Requerimiento}_n$ será insertado antes del nombre de $\texttt{Interés}_r\texttt{.Requerimiento}_s$
NEWAFT	El nombre del requerimiento $\texttt{Interés}_m\texttt{.Requerimiento}_n$ será insertado después del nombre de $\texttt{Interés}_r\texttt{.Requerimiento}_s$
NEWREP	El nombre del requerimiento $\texttt{Interés}_m\texttt{.Requerimiento}_n$ será insertado y se omite el nombre de $\texttt{Interés}_r\texttt{.Requerimiento}_s$

Cuadro 7.2 Valores de DescriptorLugar.

Cuando se utiliza BEF, AFT o REP, se crea un nuevo requerimiento compuesto y en él, se acomodan como corresponden, de manera expandida, con todo su detalle de declaraciones y condiciones, los contenidos de cada requerimiento involucrado en la regla de composición.

Cuando se utiliza NEWBEF, NEWAFT o NEWREP, se crea un nuevo requerimiento compuesto y en él, se acomodan como corresponden solamente los nombres de los requerimientos involucrados en las reglas de composición. Esto significa que solo se agregan las referencias a los requerimientos, y no todo su contenido expandido.

La regla de composición mostrada a continuación es un ejemplo del uso de AFT. El orden en el que se acomodan los elementos de los requerimientos involucrados durante el proceso de composición con cada valor aceptado de DescriptorLugar se aborda en el Capítulo 9.

```
@COMPOSITIONRULE
@ID "REGLA1";
@RULE
AgregarABitacora ORIG   @CC ObtenerConexionOk ORIG
                        @WHERE AFT
                        @CONN AND
```

Relación lógica de los requerimientos involucrados

La relación lógica de los requerimientos de una regla de composición se establece con la cláusula @CONN seguida de una de las constantes aceptadas para especificar un operador lógico. Todos los valores permitidos para los operadores lógicos se muestran en el Cuadro 7.3.

Operador lógico	Constante aceptada (OpLog)
Y	AND
O	OR
Y NO	AND_NOT
O NO	OR_NOT

Cuadro 7.3 Constantes aceptadas para los operadores lógicos.

Los operadores lógicos conectan los predicados de los esquemas del primer operando con el segundo operando. Se recomienda que los esquemas formales de los requerimientos incluidos en las reglas se encuentren normalizados antes de generar el modelo completo. El concepto de normalización se explica en el apartado 14.17.2.

En la siguiente regla de composición se muestra el uso del operador lógico `Y` con la constante `AND`.

```
@COMPOSITIONRULE
@ID "REGLA1";
@RULE
AgregarABitacora ORIG   @CC ObtenerConexionOk ORIG
                        @WHERE AFT
                        @CONN AND
```

Agregando un nuevo nombre al requerimiento compuesto

El nuevo requerimiento que se genera como producto de la regla de composición mantiene el nombre del segundo operando de la regla, a menos que se especifique otro nombre utilizando la cláusula `@NEWNAME`. El nuevo nombre debe ser válido y no debe estar en conflicto con el resto de los elementos en el proyecto actual. Un ejemplo del uso de la cláusula `@NEWNAME` se muestra a continuación. En esta regla se indica que el nombre del nuevo requerimiento compuesto será `ObtenerConexionConBitacoraNoOk`.

```
@COMPOSITIONRULE
@ID "Regla2";
@RULE
AgregarABitacora ORIG   @CC   ObtenerConexionNoOk ORIG
                        @WHERE BEF
                        @CONN AND
                        @NEWNAME "ObtenerConexionConBitacoraNoOk"
```

El uso conjunto de la cláusula `@NEWNAME` con el operador asterisco (*) agrega una consideración especial a las reglas de composición. Al utilizar el operador asterisco, se indica que un requerimiento va a afectar de la misma manera a varios más que se

encuentran localizados dentro de un mismo interés, por lo tanto al procesar esa regla, se obtendrán como resultado varios requerimientos compuestos. El uso de `@NEWNAME` en una regla que contiene el operador asterisco indica que todos los requerimientos compuestos resultantes tendrán un mismo nuevo nombre. En SPZS, el software prototipo que implementa Z-SIIMD, se ha permitido que el operador asterisco y la cláusula `@NEWNAME` puedan coexistir en una misma regla. Al nombre de cada requerimiento compuesto resultante se le agrega al final un número secuencial que lo hace diferente de los otros requerimientos que también comparten el mismo nombre nuevo introducido por la regla de composición.

Agregando nuevas declaraciones

En un requerimiento compuesto se pueden introducir nuevas declaraciones utilizando la cláusula `@ADDVAR`. Las declaraciones determinan nuevos objetos o variables y especifican su tipo. Estas nuevas declaraciones son válidas para su uso en el requerimiento compuesto; más específicamente, pueden utilizarse en la condición de composición que también se especifica en la regla mediante la cláusula `@COND`. A continuación se muestra un ejemplo del uso de la cláusula `@ADDVAR`. En esta regla, se introduce una nueva declaración en el requerimiento que será compuesto, para especificar que `numDeLinea` es un número.

```
@COMPOSITIONRULE
@ID “BitacoraAlRetiro”;
@RULE
Bitacora.AgregarABitacora ORIG
                        @CC   Retiro.* ORIG
                        @WHERE AFT
                        @CONN AND
                        @ADDVAR(
                                 “numDeLinea : Z”;
                                )
```

Agregando un predicado

Además de nuevas declaraciones, en una regla también se puede especificar un predicado para que sea incluido en el nuevo requerimiento compuesto. El predicado se introduce después de la cláusula `@COND`. A continuación se muestra un ejemplo del uso de `@COND` combinado con el uso de la cláusula `@ADDVAR` para introducir y asignar un nuevo elemento llamado `descripcion`.

```
@COMPOSITIONRULE
@ID "BitacoraAlRetiro";
@RULE
Bitacora.AgregarABitacora ORIG
                        @CC   Retiro.* ORIG
                        @WHERE AFT
                        @CONN AND
                        @ADDVAR(
                                "descripcion : RESULTADO ⟶ TEXTO";
                               )
                        @COND  (
                                "descripcion = { RetiroExitoso ↦  'Se
                                ha realizado un retiro exitosamente',
                                SaldoInsuficiente ↦  'Se ha intentado
                                retirar pero no hay saldo suficiente'
                                }";
                               )
```

Las cláusulas `@COND` y `@CONN` tienen propósitos distintos. `@CONN` únicamente presenta el operador que se va a utilizar para incorporar las condiciones de un requerimiento con las condiciones del otro, mientras que `@COND` permite agregar una cláusula adicional completa la cual puede involucrar constantes, o variables de trabajo, entrada y salida de ambos requerimientos, así como las nuevas declaraciones.

Cambiando la naturaleza de las variables

La naturaleza de una variable se refiere a su uso como variable de entrada, de salida o de trabajo. Mediante la cláusula `@CHG`, este uso puede ser modificado.

Por ejemplo, suponer que un requerimiento tiene una variable de entrada y una de salida; y que al "componerlo" con otro requerimiento, ya no es necesario que la variable de salida se considere como un resultado, sino como una variable de trabajo. En este caso, `@CHG` permite modificar la naturaleza de la variable de salida en el requerimiento original por una variable de trabajo al realizar la composición; esto es posible mediante el uso de la constante `WRKVAR`. Todos los valores aceptados para la constante `TipoVar` se encuentran listados en el Cuadro 7.4.

Tipo de variable	Constante aceptada (TipoVar)
Entrada	INVAR
Salida	OUTVAR
De trabajo	WRKVAR

Cuadro 7.4 Constantes aceptadas para los tipos de variables.

Un ejemplo de transformación de variables de entrada en variables de trabajo utilizando la constante `WRKVAR` se muestra en la siguiente regla de composición. Las variables de entrada de uno de los requerimientos pasarán a ser variables de trabajo en el requerimiento compuesto resultante.

```
@COMPOSITIONRULE
@ID "REGLA1";
@RULE
AgregarABitacora ORIG
      @CC ObtenerConexionOk ORIG
      @WHERE AFT
      @CONN AND
      @CHG (
                  NombreBitacora?=WRKVAR;
                  Fecha?=WRKVAR;
                  Actividad?=WRKVAR;
            )
```

Asignando valores a las variables

El nuevo requerimiento compuesto contendrá una serie de variables y objetos que son el resultado de combinar los elementos de ambos requerimientos involucrados usando los parámetros indicados en una regla de composición. A cualquier variable del nuevo requerimiento se le puede asignar un valor fijo mediante la cláusula `@REP`. Por ejemplo, si el requerimiento compuesto espera un valor de entrada, es posible suministrarlo directamente en el proceso de composición. La siguiente regla es un ejemplo de uso de la cláusula `@REP` para este propósito.

```
@COMPOSITIONRULE
@ID "REGLA1";
@RULE
AgregarABitacora ORIG
      @CC ObtenerConexionOk ORIG
      @WHERE AFT
      @CONN AND
      @CHG (
                  NombreBitacora?=WRKVAR;
                  Fecha?=WRKVAR;
                  Actividad?=WRKVAR;
            )
      @REP  (
                  "Fecha = 'FechaYHoraActual'";
                  "Actividad = 'Se ha obtenido
                        exitosamente una conexión del
                        pool'";
                  "NombreBitacora = 'Bitacora.log'";
            )
```

7.4. Sintaxis

En el presente apartado se encuentra la sintaxis de la extensión de Lenguaje Z propuesta por Z-SIIMD; se utiliza para este propósito la notación EBNF (*Extended Backus-Naur Form*)(ISO/IEC, 1996). Del apartado 7.4.1 al 7.4.5 se encuentra definida la sintaxis extendida para el lenguaje Z. En el apartado 7.4.6 se listan los elementos existentes en lenguaje Z tradicional que fueron usados en esta definición.

Tomando en cuenta el modelo presentado en el apartado 7.2, primero se muestra la sintaxis para el modelo base (en el apartado 7.4.1), el cual contiene las características comunes de identificación, clasificación, evolución y trazabilidad de los elementos; posteriormente se define de manera separada la sintaxis para los requerimientos (en el apartado 7.4.2), intereses (en el apartado 7.4.3), reglas de composición (en el apartado 7.4.4) y proyectos (en el apartado 7.4.5). En el apartado 7.4.6 se incluyen los elementos que se utilizan en la sintaxis de la extensión propuesta, pero que forman parte del lenguaje Z tradicional, por lo que no se profundiza en ellos. Intercalados en cada apartado se presentan diversos ejemplos de uso de esta notación; para todos los casos se van a utilizar los diferentes elementos de los casos de uso abordados en el Capítulo 11 y en el Capítulo 12.

7.4.1. Modelo base

```
ModeloBase        =     '@ID',  NombreId,
                        ['@DESCRIPTION', mensaje],
                        ['@CLASS', '@CAT', term, {term}, '@PAL', term,
                        {term}],
                        ['@EVOL', regEvol, {regEvol}],
                        ['@TRAZ', regTraz, {regTraz}],
                        puntoYComa;

term              =     mensaje, puntoYComa;

regEvol           =     '@ID', mensaje,
                        '@TYPE', tipoEvol,
                        '@DATE', mensaje,
                        '@TIME', mensaje,
                        '@ELEMENT', mensaje,
                        '@BEFORE', mensaje,
                        '@AFTER', mensaje,
                        puntoYComa;

regTraz           =     '@DATE', mensaje,
                        '@TIME', mensaje,
                        '@ACTIVITY', acts,
                        '@MAP', maps,
                        '@COMMENT', mensaje,
                        puntoYComa;
```

```
tipoEvol          =     'AGREGADO' | 'BORRADO' | 'EDITADO' | mensaje;

maps              =     'CARACTERISTICA' | 'DECISION' | 'ASPECTO' |
                        mensaje;

acts              =     'ESPECIFICACION' | 'ARQUITECTURA' | 'DISEÑO' |
                        'IMPLEMENTACION' | 'PRUEBA' | 'EVOLUCION' |
                        mensaje;

mensaje           =     comillas, cadena, comillas;

puntoYComa        =     ';';

NombreId          =     comillas, identificador, comillas;
```

7.4.2. Requerimiento

```
Requerimiento     =     '@REQUIREMENT', ModeloBase,
                        '@FORMAL', repFormal;

repFormal         =     mensaje, puntoYComa;
```

Ejemplo

Como ejemplo de la especificación de un requerimiento se mostrará la definición de los dos requerimientos relacionados con la especificación para agregar un registro a una bitácora.
El primer requerimiento es identificado como "*TiposUtilizados*" y solo contiene los tipos necesarios para la especificación:

```
@REQUIREMENT
@ID "TiposUtilizados"
@DESCRIPTION "Definición de los tipos utilizados en la
bitácora";
@FORMAL
"SCHCHAR TiposUtilizados
[TEXTO]
BITACORA == ℙ (TEXTO × TEXTO)
ENDCHAR";
```

El segundo requerimiento se ha identificado como "*AgregarABitacora*" y es el que contiene la especificación para insertar una nueva entrada en la bitácora:

```
@REQUIREMENT
@ID "AgregarABitacora"
@DESCRIPTION "Insertar un registro en la bitácora";
@FORMAL
"SCHCHAR AgregarABitacora
NombreBitacora? : TEXTO
Fecha? : TEXTO
         Actividad? : TEXTO
a! : BITACORA
obtenerBitacora: TEXTO ⤖ BITACORA
|
∃1 x:BITACORA  • x = obtenerBitacora( NombreBitacora? ) ^  x ≠ ∅
a!  =  obtenerBitacora( NombreBitacora? )  ∪ {Fecha? ↦ Actividad? }
ENDCHAR";
```

7.4.3. Interés

```
Interes          =     '@CONCERN', ModeloBase,
                       '@REQS', Tr;

Tr               =     (NombreId, puntoYComa |Requerimiento, puntoYComa),
                       {NombreId, puntoYComa |Requerimiento, puntoYComa};
```

Ejemplo

Para ejemplificar la especificación de un interés, se tomará el mismo concepto de la bitácora abordado anteriormente. Este interés es nombrado como "*Bitacora*" y agrupa los dos requerimientos previamente especificados en el ejemplo del apartado 7.4.2; además se clasifica con dos categorías: "almacenamiento" y "no funcional", y también se etiqueta con tres palabras clave: "registro", "bitacora" y "bitácora".

```
@CONCERN
@ID "Bitacora"
@DESCRIPTION "Requerimientos para implementar una
bitacora"
@CLASS
@CAT "almacenamiento"; "no funcional";
@PAL "registro"; "bitacora"; "bitácora";;
@REQS "TiposUtilizados"; "AgregarABitacora";
```

7.4.4. Regla de composición

```
Regla            =     '@COMPOSITIONRULE', ModeloBase,
                       '@RULE', Regla1;

Regla1           =     reqId1, ver, '@CC', (reqId1 | reqId2), ver,
                       '@WHERE', descriptorLugar,
                       '@CONN', opLog,
                       ['@NEWNAME', NombreId,],
                       ['@ADDVAR', addCambios,],
                       ['@COND', condCambios,],
                       ['@CHG', chgCambios,],
                       ['@REP', repCambios];

reqId1           =     identificador |
                       identificador, punto, identificador;

reqId2           =     identificador, punto, asterisco;

ver              =     'ORIG' | 'COMP';

descriptorLugar  =     'BEF' | 'AFT' | 'REP' |
                       'NEWBEF' | 'NEWAFT' | 'NEWREP';

opLog            =     'AND' | 'OR' | 'ANDNOT' | 'ORNOT';

condCambios      =     parentesis1, mensaje, puntoYComa,
                       {mensaje, puntoYComa}, parenteis2;

addCambios       =     parentesis1, mensaje, puntoYComa,
                       {mensaje, puntoYComa}, parentesis2;

chgCambios       =     parentesis1, chgCam, puntoYComa,
                       {chgCam, puntoYComa}, parentesis2;

chgCam           =     identificador, igual,  nat;

nat              =     'WRKVAR' | 'INVAR' | 'OUTVAR';

repCambios       =     parentesis1, mensaje, puntoYComa,
                       {mensaje, puntoYComa}, parentesis2;

asterisco        =     '*';

comillas         =     '"';

punto            =     '.';

igual            =     '=';

parentesis1      =     '(';

parentesis2      =     ')';
```

Ejemplo

El siguiente ejemplo muestra la especificación de una regla de composición que no involucra intereses. El identificador de la regla es "*Regla1*". El primer requerimiento que interviene se llama "*AgregarABitacora*" y el segundo requerimiento se llama "*ObtenerConexionOk*". En esta regla se realizan cambios en la naturaleza de tres variables y se realizan reemplazos de valores específicos para ellas.

```
@COMPOSITIONRULE
@ID "REGLA1";
@RULE
AgregarABitacora ORIG
      @CC ObtenerConexionOk ORIG
      @WHERE AFT
      @CONN AND
      @CHG (
                  NombreBitacora?=WRKVAR;
                  Fecha?=WRKVAR;
                  Actividad?=WRKVAR;
            )
      @REP  (
                  "Fecha = 'FechaYHoraActual'";
                  "Actividad = 'Se ha obtenido
                        exitosamente una conexión del
                        pool'";
                  "NombreBitacora = 'Bitacora.log'";

            )
```

7.4.5. Proyecto

```
Proyecto          =     '@PROJECT',  ModeloBase,
                        '@CONCERNS', tempCon,
                        '@REQUIREMENTS', tempReq,
                        '@RULES',    tempRule;

tempCon           =     {NombreId, puntoYComa |Interes, puntoYComa};

tempReq           =     {NombreId, puntoYComa |Requerimiento, puntoYComa};

tempRule          =     {NombreId, puntoYComa |Regla, puntoYComa};
```

Ejemplo

Un proyecto con nombre "*Proyecto1*", que agrupa los dos intereses "*Bitacora*" y "*ObtenerConexionOk*", así como la regla de composición denominada "*Regla1*" queda especificado de la siguiente manera:

```
@PROJECT
@ID "Proyecto1";
@CONCERNS "Bitacora"; "ObtenerConexionOk";
@REQUIREMENTS
@RULES "Regla1";
```

7.4.6. Símbolos definidos en Z tradicional

La siguiente es una lista de símbolos terminales utilizados en la presente gramática, pero no definidos aquí, pues son parte del Lenguaje Z tradicional. Se puede consultar (ISO/IEC-13568, 2002) y (SPIVEY, J. M., 1992).

```
identificador     Identificador válido en Lenguaje Z

cadena            Cadena de caracteres válida en Lenguaje Z
```

7.5. Conclusiones

Resulta imprescindible conocer la representación de requerimientos, intereses, reglas y proyectos para poder entender y trabajar con los procesos de composición y reutilización que se abordarán en los siguientes apartados. En este capítulo, además de mostrar los modelos de cada uno de estos elementos, se explicaron también las posibilidades de configuración soportadas por las reglas y se presentó una sintaxis para la definición formal de la extensión de Z-SIIMD.

Capítulo 8.
Modelo de composición

8.1. Introducción

En este capítulo se abordan los pormenores del modelo de composición de Z-SIIMD. El proceso de composición representa un elemento fundamental del presente trabajo de tesis, pues hace posible la obtención de un modelo completo que unifica todos los requerimientos de acuerdo a las relaciones configuradas en la especificación de las reglas.

Como resultado final de este proceso se obtiene un modelo compuesto, el cual puede ser producido en cualquier momento a partir de un conjunto válido de reglas de composición y requerimientos. El modelo compuesto registra la influencia de unos requerimientos sobre otros y proporciona versiones finales de las especificaciones originales capturadas en un proyecto.

La explicación del modelo de composición se encuentra en el apartado 8.2; en el apartado 8.3 se aborda el algoritmo de composición, primero, desde una perspectiva de un diagrama de alto nivel, y después, con un pseudocódigo con mayor nivel de detalle. También en ese apartado se listan las principales subrutinas utilizadas en el algoritmo, incluyendo una descripción general de las tareas que realizan.

8.2. Modelo de composición

La composición es el proceso que genera un modelo integrado y expandido que resulta de aplicar las instrucciones de las reglas de composición a los requerimientos involucrados. Este proceso toma como entradas, las especificaciones originales y el conjunto de reglas para construir por partes cada uno de los requerimientos compuestos, y posteriormente, entregar un modelo final completo (Figura 8.1). La construcción unitaria de cada requerimiento compuesto es incremental: cada paso va agregando nuevos elementos de contenido al resultado final dependiendo de la configuración de los valores de los parámetros de las reglas.

El conjunto de requerimientos afectados por las reglas se encuentra contenido en los requerimientos compuestos. Por otra parte, el modelo final completo es el conjunto de requerimientos, compuestos o no, que conforman todo el proyecto. Este modelo agrupa la totalidad de las especificaciones del proyecto en sus versiones finales. Es posible que en un proyecto existan requerimientos que no estén involucrados en las reglas de composición; el conjunto de requerimientos compuestos no los incluirá, pero sí serán agregados al modelo final completo.

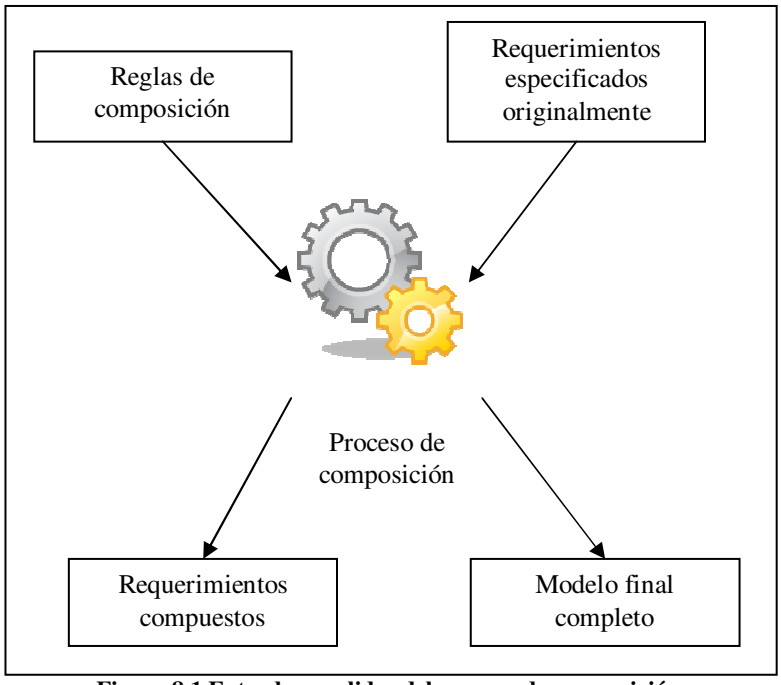


Figura 8.1 Entradas y salidas del proceso de composición.

Tanto el modelo final completo como el conjunto de requerimientos compuestos son generados como productos del algoritmo de composición. Cada vez que el algoritmo se ejecuta, se toman como entrada del proceso, las reglas y requerimientos contenidos en el proyecto en ese momento; de tal manera que cada cambio realizado en la especificación original de requerimientos o reglas, impacta el contenido generado de ambos productos.

Las reglas son la guía para el proceso de composición y determinan la estructura individual de los requerimientos compuestos; cada regla contiene una serie de parámetros que indican la configuración del requerimiento final para cada requerimiento impacta a otros. El proceso de composición de dos requerimientos que intervienen en una regla se muestra en la Figura 8.2.

Es importante mencionar que el proceso de composición no es conmutativo. Esto quiere decir que el modelo final obtenido puede no ser el mismo si se invierte el orden de los requerimientos involucrados en una regla de composición. Esto no debe suponer un mayor problema si se tienen bien identificados el requerimiento transversal y el requerimiento que resulta atravesado. En una regla de composición, el requerimiento transversal siempre aparece como "Requerimiento 1" y el atravesado como "Requerimiento 2". Una forma de leer esta relación sería: "El 'Requerimiento 1' atraviesa (o impacta, o afecta) al 'Requerimiento 2'". Cuando se tiene clara esta relación, invertir el orden carece de sentido.

Por otra parte, si los requerimientos de un proyecto están agrupados en intereses, una regla puede especificar que un requerimiento afecta a varios localizados dentro de un mismo

interés, tal como se muestra en la Figura 8.3. En ese caso, todos los valores de los parámetros que contiene la regla se aplican a la composición de cada requerimiento afectado. Es decir, esto equivale a escribir reglas individuales para cada uno de los requerimientos afectados; todas con los mismos parámetros de configuración. Este escenario es posible siempre y cuando en el proyecto se cuente con una organización estratégica de intereses y requerimientos.

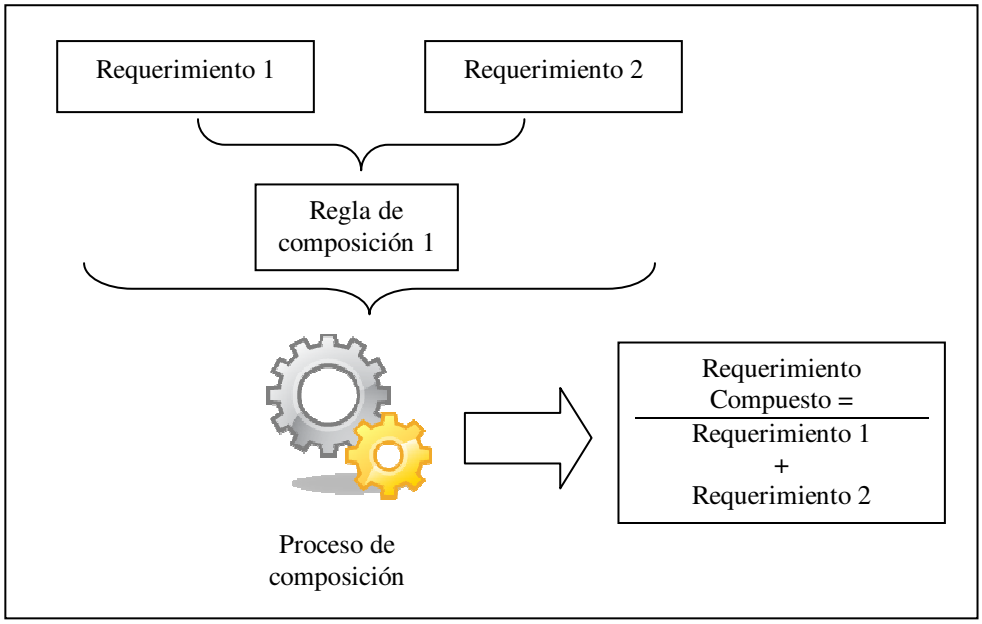


Figura 8.2 Composición de dos requerimientos involucrados en una regla.

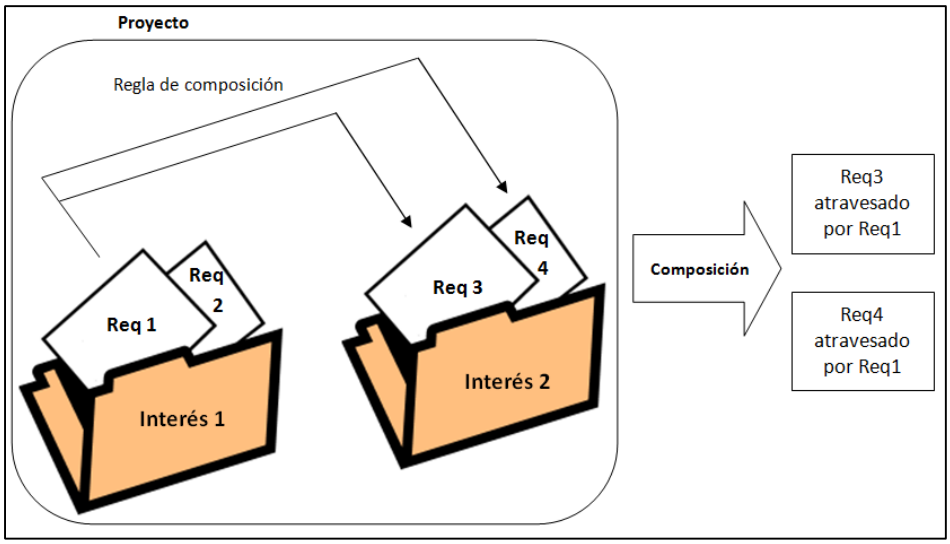


Figura 8.3 Varios requerimientos afectados por la misma regla de composición.

8.3. Algoritmo de composición

8.3.1. Objetivo del algoritmo

En la Figura 8.4 se muestra el algoritmo del proceso de composición, el cual gestiona dos estructuras en memoria: una donde se almacenan solo los requerimientos compuestos, y otra que contiene el modelo completo a partir de los requerimientos originales y los compuestos.

En el proceso de creación de la estructura que contiene los requerimientos compuestos, se realizan dos iteraciones: una para recorrer cada una de las reglas de composición, pues son ellas las que trazan la directriz para los requerimientos compuestos, y otra para recorrer cada uno de los requerimientos involucrados como segundo operador en cada regla, pues la regla puede incluir un asterisco en el segundo operador, indicando que son varios los requerimientos atravesados que esa regla produce.

Como ya se ha mencionado, en una regla de composición, un requerimiento únicamente puede ser atravesado por otro requerimiento a la vez. Sin embargo, reglas distintas pueden afectar al mismo requerimiento en diferentes momentos haciendo que éste sea atravesado varias veces (una vez por cada regla de composición que lo incluya como segundo operador). En una regla se puede seleccionar la versión del requerimiento con la que se desea trabajar en el proceso de composición.

La idea central del algoritmo es crear por partes cada uno de los requerimientos compuestos de acuerdo a los parámetros de composición. Es decir, se trata de construir incrementalmente con acciones separadas el contenido de cada sección de los requerimientos según se indique en las cláusulas `@WHERE`, `@CONN`, `@COND`, `@NEWNAME`, `@ADD`, `@CHG`, `@REP` de cada regla.

Una vez que el proceso de construcciones individuales ha terminado, se crea el modelo completo que contiene todos los requerimientos del sistema con las afectaciones que la composición les haya provocado. Para esto, se toma en cuenta que hay requerimientos que se generaron con un nuevo identificador, y además, que no todos los requerimientos especificados inicialmente se incluyen en las reglas de composición.

Una vez que el algoritmo ha producido un modelo que contiene todos los requerimientos originales y compuestos, la herramienta de software desarrollada (en este caso SPZS) debería ofrecer la posibilidad de elegir una vista reducida que corresponda a los requerimientos con los que se desea trabajar. Algunas opciones pueden ser:

- Todos los requerimientos compuestos y no compuestos.
- Solamente los requerimientos compuestos.
- Solamente los requerimientos especificados originalmente.
- Solamente los requerimientos compuestos con nuevo nombre.

Estas opciones fueron justamente las que se implementaron en SPZS para el presente trabajo; sus características en detalle se presentan en el Capítulo 10.

Es así como utilizando diferentes vistas en una herramienta de software, la organización original de los intereses, requerimientos y reglas de composición se mantiene intacta mientras se permite el acceso al modelo compuesto.

8.3.2. Diagrama de alto nivel

La Figura 8.4 muestra el algoritmo de composición desde una perspectiva general que cubre las acciones más relevantes llevadas a cabo por el proceso de composición.

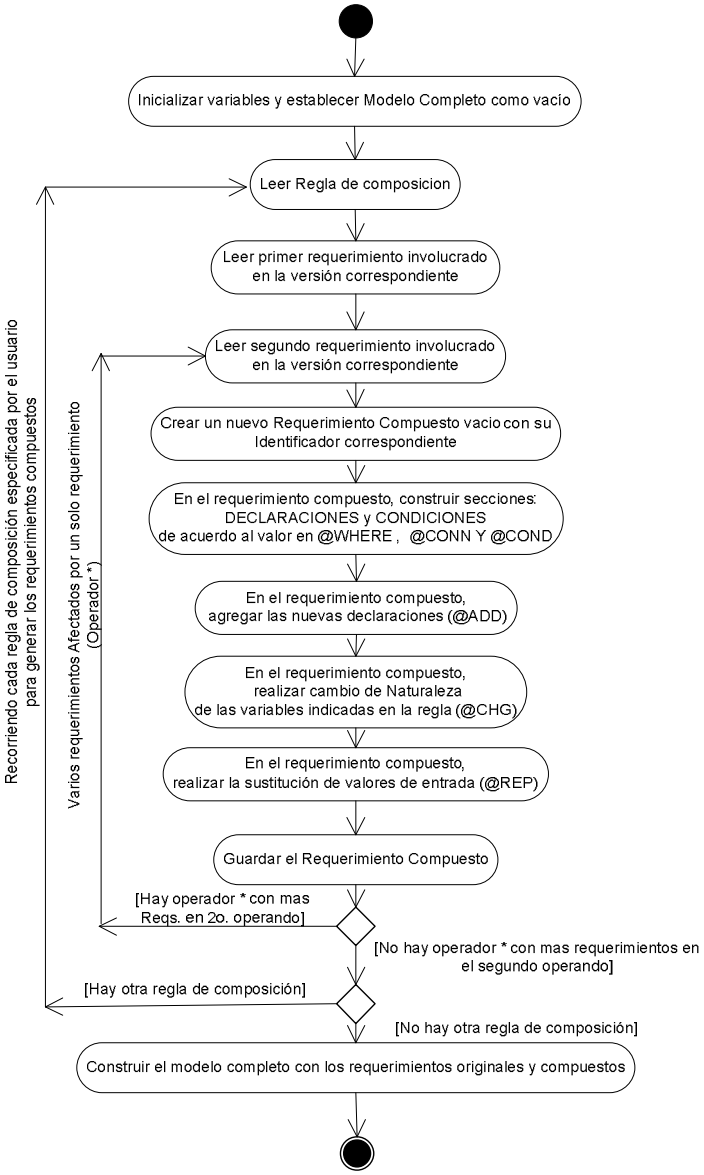


Figura 8.4 Algoritmo de alto nivel para el proceso de composición.

8.3.3. Pseudocódigo

En este apartado, en la Figura 8.5 se muestra un pseudocódigo más detallado para el algoritmo de composición del diagrama de la Figura 8.4. La respectiva explicación se encuentra al final del cuerpo del pseudocódigo. La Figura 8.6 muestra las principales subrutinas que se utilizaron en el algoritmo y en la Figura 8.7 se mencionan las subrutinas secundarias.

```
Nombre de la rutina:  COMPONER
Descripción:          Algoritmo de composición de requerimientos mediante reglas.
Entradas:             Conjunto de reglas de composición (ReglasComposicion) y el conjunto de
                      requerimientos especificados originalmente (Requerimientos).
Salidas:              ReqCompuestos (Estructura que contiene solo los requerimientos
                      afectados por las reglas de composición)
                      ModeloCompleto (Estructura que contiene todos los requerimientos de la
                      especificación).
Inicio-Rutina
    /* Definiendo los tipos utilizados: */
    /* El uso de [] indica una estructura con varios elementos iguales */
    1)  Req1, Req2, ReqCompuesto : REQ
    2)  ReglaComp : REGLA
    3)  ReglasComposicion: REGLA[]
    4)  Requerimientos, ReqCompuestos, ModeloCompleto : REQ[]

    /* Inicia el algoritmo: */
    5)  Inicializar las estructuras:  ReqCompuestos = Vacío   y    ModeloCompleto = Vacío
    6)  Iterar Para cada regla de composición ReglaComp en ReglasComposicion:
        6.1)    Leer ReglaComp
        6.2)    Leer Req1 y Req2 involucrados en ReglaComp
        6.3)    Si HayErrores(Req1)=Si  O  HayErrores(Req2)=Si  O  HayErrores(ReglaComp)=Si
                        IndicarError ("Error en los requerimientos y/o en la regla de
                        composición");
                        Salir de la Iteración
                Fin-Si
        6.4)    Si ReqCompuestos.Existe (Req1) = Si  Y  ReglaComp.VerReq1 = "COMP"
                        Req1 = ReqCompuestos.Traer(Req1)
                Fin-Si
        6.5)    Iterar para cada Requerimiento Req2 que involucre el segundo operador de la
                regla (Por Si incluye *)
                6.5.1)  ReqCompuesto = Nuevo Requerimiento Compuesto vacío.
                6.5.2)   Si ReqCompuestos.Existe(Req2) = Si  Y  ReglaComp.VerReq2 = "COMP"
                                Req2 = ReqCompuestos.Traer(Req2)
                        Fin-Si
                6.5.3)  EnCaso (ReglaComp.DescriptorLugar)
                        Inicio-EnCaso
                                "AFT" : COMPOSICION_AFT(Req1, Req2, ReqCompuesto, ReglaComp)
                                "BEF" :  COMPOSICION_BEF(Req1, Req2, ReqCompuesto, ReglaComp)
                                "REP" :  COMPOSICION_REP(Req1, ReqCompuesto, ReglaComp)
                                "NEWAFT": COMPOSICION_NEWAFT(Req1, Req2, ReqCompuesto,
                                                                        ReglaComp)
                                "NEWBEF": COMPOSICION_NEWBEF(Req1, Req2, ReqCompuesto,
                                                                        ReglaComp)
                                "NEWREP": COMPOSICION_NEWREP(Req1, ReqCompuesto, ReglaComp)
                        Fin-EnCaso
                6.5.4)  CambiarNaturalezaVariables (ReqCompuesto, ReglaComp)
                6.5.5)  CambiarValoresEntrada (ReqCompuesto, ReglaComp)
                6.5.6)  Si HayErrores (ReqCompuesto) = No
                                6.5.6.1) Si ReqCompuestos.Existe (ReqCompuesto) = Si
                                                ReqCompuestos.Reemplazar (ReqCompuesto)
                                        Si-No
                                                ReqCompuestos.Agregar(ReqCompuesto)
                                        Fin-Si
                        Si-No
                                6.5.6.2) IndicarError ("Error en el requerimiento compuesto");
                        Fin-Si
            Fin-Iteración
        Fin-Iteración
    7)  Iterar para cada Requerimiento Req registrado en el sistema (Originales y Compuestos):
                7.1)    Si ReqCompuestos.Existe (Req) = Si
                                ModeloCompleto.ActualizarCon (ReqCompuestos.Req)
                        Si-No
                                ModeloCompleto.ActualizarCon (Req)
                        Fin-Si
        Fin- Iteración
Fin-Rutina
```

Figura 8.5 Pseudocódigo para el proceso de composición.

- Pasos del 1 al 4: se realiza la declaración de las variables. *REQ* indica un único requerimiento y *REGLA* indica una sola regla, mientras que *REQ[]* hace referencia a una colección de requerimientos y *REGLA[]* se refiere a una colección de reglas.

- Paso 5: se inicializa *ReqCompuestos*,. Es importante mencionar que *ReqCompuestos* es el conjunto de requerimientos ya compuestos que resultan al aplicar las reglas; mientras que *ModeloCompleto* es una estructura que al final del proceso de composición contiene todos los requerimientos (compuestos y/o no compuestos) que describen en su totalidad el sistema especificado.

- Paso 6: Todas las actividades incluidas en el paso 6 se deben repetir para cada regla de composición que existe en el modelo.

  - Paso 6.1: se lee la regla de composición y en el paso 6.2 se leen los dos requerimientos involucrados en esa regla. Éstos son los datos clave que el algoritmo utiliza para generar los resultados.

  - Paso 6.2: se verifica que los requerimientos y las reglas de composición se encuentren bien formados y sin errores. Si se detecta algún error, el proceso de composición se detiene.

  - Paso 6.3: se verifica que ni la regla ni los requerimientos involucrados contengan errores.

  - Paso 6.4: se verifica la posibilidad de que en la regla se haya especificado tomar la versión compuesta más reciente del primer requerimiento y además, que ya exista en los requerimientos compuestos una versión atravesada de él; de ser así, se debe tomar la versión compuesta existente y no el requerimiento inicialmente especificado por el usuario.

  - Paso 6.5: se considera la posible inclusión del operador asterisco (*) para denotar más de un requerimiento en el segundo operando de la regla. En este caso se debe realizar una iteración sobre todos los requerimientos que se incluyan en el interés que aparezca como segundo operador en la regla.

    - Paso 6.5.1: se inicializa el Requerimiento Compuesto (a un estado vacío) pues el algoritmo lo irá construyendo poco a poco durante el resto de los pasos.

    - Paso 6.5.2: se hace lo mismo que el paso 6.4, pero para el segundo requerimiento que involucra la regla.

    - Paso 6.5.3: el requerimiento compuesto se comienza a construir por partes. El orden del contenido de las dos principales secciones: Declaraciones y Condiciones, está determinado por `DescriptorLugar`, tal como se indica en el Cuadro 8.1. La notación utilizada se muestra en el Cuadro 8.2.

    - Paso 6.5.4: se procede a realizar el cambio de naturaleza de las variables que fue estipulado en la regla de composición (Por ejemplo, variables de salida que al componerse dejan de serlo para convertirse en variables de trabajo).

- Paso 6.5.5: se encarga de colocar valores fijos a variables que se encuentran en los requerimientos involucrados (Por ejemplo, hacer que una variable de entrada tome siempre un valor constante).

- Paso 6.5.6: se realiza una verificación de posibles errores que hayan aparecido en el requerimiento recién compuesto durante el proceso.

    - Paso 6.5.6.1: si no se detectaron errores, se busca una versión previa de este mismo requerimiento en *ReqCompuestos* (existiría otra versión más antigua en el caso de que este mismo requerimiento haya sido afectado por otros anteriormente como resultado de el efecto de otras reglas). Si se encuentra una versión anterior, la versión actual del requerimiento compuesto debe reemplazar a la antigua; si no, simplemente se debe agregar al modelo completo.

- Pasos 7 y 7.1: Una vez que todos los requerimientos han sido compuestos, se comienza a construir el modelo completo. En este paso se determina si se trata de un requerimiento compuesto, en cuyo caso se extrae de la estructura *ReqCompuestos*, o si se trata de uno que no formó parte de ninguna regla de composición, en cuyo caso se extrae de su versión aislada original especificada por el usuario.

Valor de DescriptorLugar en la regla de composición	Requerimiento Compuesto	
	Secuencia en la sección "Declaraciones"	Secuencia en la sección "Invariantes"
AFT	1) Req2.Declaraciones 2) Req1.Declaraciones 3) ReglaComp.AddVar	1) Req2.Condiciones 2) ReglaComp.OpLog 3) Req1.Condiciones 4) ReglaComp.Rep 5) ReglaComp.Cond
BEF	1) Req1.Declaraciones 2) Req2.Declaraciones 3) ReglaComp.AddVar	1) Req1.Condiciones 2) ReglaComp.OpLog 3) Req2.Condiciones 4) ReglaComp.Rep 5) ReglaComp.Cond
REP	1) Req1.Declaraciones 2) ReglaComp.AddVar	1) Req1.Condiciones 2) ReglaComp.Rep 3) ReglaComp.Cond
NEWAFT	1) Req2.Identificador 2) Req1.Identificador 3) ReglaComp.AddVar	1) ReglaComp.Rep 2) ReglaComp.Cond
NEWBEF	1) Req1.Identificador 2) Req2.Identificador 3) ReglaComp.AddVar	1) ReglaComp.Rep 2) ReglaComp.Cond
NEWREP	1) Req1.Identificador 2) ReglaComp.AddVar	1) ReglaComp.Rep 2) ReglaComp.Cond

Cuadro 8.1 Valores de DescriptorLugar y secuencia en las secciones de declaraciones e invariantes.

Elemento	Significado
Req1.Identificador	Identificador del requerimiento indicado en la regla de composición en el primer operando.
Req1.Declaraciones	Declaraciones del requerimiento indicado en la regla de composición en el primer operando.
Req1.Condiciones	Condiciones del requerimiento indicado en la regla de composición en el primer operando.
Req2.Identificador	Identificador del requerimiento indicado en la regla de composición en el segundo operando.
Req2.Declaraciones	Declaraciones del requerimiento indicado en la regla de composición en el segundo operando.
Req2.Declaraciones	Condiciones del requerimiento indicado en la regla de composición en el segundo operando.
ReglaComp.OpLog	Operador lógico indicado en la regla de composición.
ReglaComp.AddVar	Declaraciones adicionales indicadas en la regla para incluirse en el requerimiento compuesto.
ReglaComp.Cond	Condición adicional indicada en la regla para incluirse en el requerimiento compuesto.
ReglaComp.Rep	Valores para reemplazar indicados en la regla de composición.

Cuadro 8.2 Notación utilizada en el Cuadro 8.1.

```
Nombre de la subrutina: COMPOSICION_AFT
Descripción: Crea el Requerimiento compuesto cuando DescriptorLugar = "AFT"
Datos de entrada (Por referencia): Req1, Req2, ReqCompuesto, ReglaComp
Datos de salida: Ninguno
Inicio-Subrutina
        ReqCompuesto.Declaraciones = Req2.Declaraciones + Req1.Declaraciones +
                                                                ReglaComp.AddVar
        ReqCompuesto.Condiciones = Req2.Condiciones + ReglaComp.OpLog + Req1.Condiciones +
                                                                     ReglaComp.Cond
Fin-Subrutina

Nombre de la subrutina: COMPOSICION_BEF
Descripción: Crea el Requerimiento compuesto cuando DescriptorLugar = "BEF"
Datos de entrada (Por referencia): Req1, Req2, ReqCompuesto, ReglaComp
Datos de salida: Ninguno
Inicio-Subrutina
        ReqCompuesto.Declaraciones =  Req1.Declaraciones + Req2.Declaraciones +
                                                                ReglaComp.AddVar
        ReqCompuesto.Condiciones = Req1.Condiciones + ReglaComp.OpLog + Req2.Condiciones +
                                                                     ReglaComp.Cond
Fin-Subrutina

Nombre de la subrutina: COMPOSICION_REP
Descripción: Crea el Requerimiento compuesto cuando DescriptorLugar = "REP"
Datos de entrada (Por referencia): Req1, ReqCompuesto, ReglaComp
Datos de salida: Ninguno
Inicio-Subrutina
        ReqCompuesto.Declaraciones = Req1.Declaraciones + ReglaComp.AddVar
        ReqCompuesto.Condiciones = Req1.Condiciones + ReglaComp.Cond
Fin-Subrutina

Nombre de la subrutina: COMPOSICION_NEWAFT
Descripción: Crea el Requerimiento compuesto cuando DescriptorLugar = "NEWAFT"
Datos de entrada (Por referencia): Req1, Req2, ReqCompuesto, ReglaComp
Datos de salida: Ninguno
Inicio-Subrutina
        ReqCompuesto.Declaraciones = Req2.Identificador + Req1.Identificador +
                                                                     ReglaComp.AddVar
        ReqCompuesto.Condiciones = ReglaComp.Cond
Fin-Subrutina

Nombre de la subrutina: COMPOSICION_NEWBEF
Descripción: Crea el Requerimiento compuesto cuando DescriptorLugar = "NEWBEF"
Datos de entrada (Por referencia): Req1, Req2, ReqCompuesto, ReglaComp
Datos de salida: Ninguno
Inicio-Subrutina
        ReqCompuesto.Declaraciones = Req1.Identificador + Req2.Identificador +
                                                                ReglaComp.AddVar
        ReqCompuesto.Condiciones = ReglaComp.Cond
Fin-Subrutina

Nombre de la subrutina: COMPOSICION_NEWREP
Descripción: Crea el Requerimiento compuesto cuando DescriptorLugar = "NEWREP"
Datos de entrada (Por referencia): Req1, ReqCompuesto, ReglaComp
Datos de salida: Ninguno
Inicio-Subrutina
        ReqCompuesto.Declaraciones = Req1.Identificador + ReglaComp.AddVar
        ReqCompuesto.Condiciones = ReglaComp.Cond
Fin-Subrutina
```

Figura 8.6 Principales subrutinas utilizadas en el pseudocódigo del proceso de composición.

Nombre de la subrutina: **CambiarNaturalezaVariables**
Descripción: Se encarga de realizar los cambios de naturaleza en las variables del requerimiento compuesto que se han indicado en la regla de composición.
Datos de entrada: Regla de composición, Requerimiento compuesto
Datos de salida: Ninguna

Nombre de la subrutina: **CambiarValoresEntrada**
Descripción: Se encarga de realizar los cambios de las variables por los valores indicados en la regla de composición.
Datos de entrada: Regla de composición, Requerimiento compuesto
Datos de salida: Ninguna

Nombre de la subrutina: **HayErrores**
Descripción: Se encarga de verificar que un requerimiento esté correctamente formado. Es decir, que no existan errores Sintácticos o Semánticos, y que sus componentes sean válidos.
Datos de entrada: Una Regla de composición
Datos de salida: Valor lógico (SI o NO).

Nombre de la subrutina: **ReqCompuestos.Existe**
Descripción: Determina si el requerimiento indicado ya ha sido agregado en el modelo completo.
Datos de entrada: Un Requerimiento
Datos de salida: Valor lógico (SI o NO).

Nombre de la subrutina: **ReqCompuestos.Reemplazar**
Descripción: Se encarga de reemplazar una versión anterior de un requerimiento compuesto por una versión más nueva, que ha resultado de una nueva composición (más de un requerimiento lo ha atravesado).
Datos de entrada: Un Requerimiento
Datos de salida: Ninguno

Nombre de la subrutina: **ReqCompuestos.Agregar**
Descripción: Agrega el requerimiento indicado al conjunto de requerimientos que forman el modelo completo actual.
Datos de entrada: Un requerimiento
Datos de salida: Ninguno

Nombre de la subrutina: **IndicarError**
Descripción: Se encarga de registrar un error que ha ocurrido ya sea para indicarlo al usuario o para proceder en consecuencia.
Datos de entrada: Error Ocurrido
Datos de salida: Ninguno

Nombre de la subrutina: **Modelocompleto.ActualizarCon**
Descripción: Se encarga de actualizar el modelo completo con los datos del requerimiento que se suministra como entrada.
Datos de entrada: Un requerimiento
Datos de salida: Ninguno

Figura 8.7 Otras subrutinas utilizadas en el pseudocódigo del proceso de composición.

8.4. Conclusiones

Las reglas son las estructuras que contienen las instrucciones para guiar el proceso de composición, el cual se encarga de tomar los requerimientos especificados originalmente y las reglas introducidas para producir requerimientos compuestos a partir de esas entradas. Los productos del proceso de composición son generados por un algoritmo, por lo que cualquier cambio en las reglas o en los requerimientos especificados originalmente impactará los resultados que se obtienen como resultado del proceso de composición.

Por otra parte, es importante mencionar que el algoritmo de composición fue probado con proyectos con muy pocos requerimientos (20 aproximadamente como promedio). De acuerdo a la clasificación mostrada en (JIANG, Li et al., 2005), los proyectos pueden ser: muy pequeños (si tienen menos de 100 requerimientos), pequeños (si tienen entre 100 y 500 requerimientos), medianos (si tienen entre 500 y 1000 requerimientos), grandes (si poseen entre 1000 y 4000 requerimientos), o muy grandes (si su número de requerimientos es mayor de 4000). De esta manera, el algoritmo de composición fue probado con proyectos muy pequeños; quedando como trabajo futuro la comprobación del algoritmo en proyectos más grandes. Por otra parte, también queda como trabajo futuro medir y analizar la complejidad del algoritmo para mejorar su eficiencia.

.

Capítulo 9.
Modelo de reutilización

9.1. Introducción

En este capítulo se aborda el proceso de reutilización, el cual es el encargado de gestionar el acceso a los repositorios. Los repositorios son almacenes de elementos modulares que han sido seleccionados estratégicamente de proyectos previamente especificados con el objetivo de que sean reutilizados en otros proyectos. Estos almacenes están disponibles desde cualquier especificación, por lo tanto, en cualquier momento se pueden agregar elementos de un proyecto al repositorio, o viceversa. Existen tres repositorios, uno para cada tipo de los elementos que se pueden reutilizar: requerimientos, intereses y reglas de composición.

Entre las principales funciones del modelo de reutilización se encuentran el almacenamiento y la búsqueda de elementos para su instanciación en un proyecto. El almacenamiento se refiere a la incorporación de elementos de un proyecto al repositorio. La búsqueda se refiere a utilizar un criterio para obtener una lista de elementos existentes que cumplan con él. La instanciación es el proceso de incorporar al proyecto actual, un elemento del repositorio.

Las dos funciones: clasificación e instanciación se exponen separadamente ya que se llevan a cabo en dos momentos diferentes. En el apartado 9.2 se explica el modelo de reutilización. El apartado 9.3 aborda el algoritmo de reutilización; se incluye un diagrama de alto nivel y también un pseudocódigo con mayor nivel de detalle. También se listan las principales subrutinas utilizadas en los algoritmos, incluyendo una descripción general de las tareas que realizan.

9.2. Modelo de reutilización

Los elementos que pueden ser reutilizados e instanciados son requerimientos, intereses y reglas de composición. En general estos elementos son susceptibles de reutilizarse si se identifica la posibilidad de incorporarlos en otros proyectos diferentes al que pertenecen. Su significado debe ser tal, que deben poder aplicarse en diferentes contextos manteniendo su estructura y contenido.

Es recomendable que antes de reutilizar cualquier elemento, primero sea evaluado para garantizar el cumplimiento de ciertos parámetros de calidad. Esto con la finalidad de que, al ser instanciado en otro proyecto, los errores contenidos en él no se propaguen y la calidad del sistema no se vea comprometida. Por ejemplo, aunque técnicamente es posible reutilizar cualquier requerimiento, no es deseable que se almacenen requerimientos

ambiguos o incompletos, aunque en nuestro caso la ambigüedad queda eliminada al utilizar notación formal. Los criterios de calidad que se establezcan están más allá del alcance del presente trabajo, y deberán ser considerados por el equipo de desarrollo de cada proyecto.

Otro punto que se debe considerar para determinar la pertinencia de la reutilización de elementos es el nivel de abstracción con el cual se especifican. Si los elementos han sido capturados en un nivel similar y compatible de especificación, en donde sus entradas y salidas coinciden y tienen sentido con las entradas y salidas del resto de los elementos de ese proyecto, puede pensarse en una reutilización factible.

9.2.1. Clasificación

Se pueden mencionar algunos otros criterios indicadores que motivan la reutilización de los elementos desde la perspectiva particular de cada uno de ellos; sus características individuales permiten vislumbrar distintas áreas de oportunidad para llevar a cabo la reutilización de requerimientos, intereses y reglas de composición.

Los requerimientos son la unidad mínima de especificación que puede ser reutilizada sin dependencias. Los intereses, al ser agrupamientos de requerimientos, no pueden concebirse sin ellos, ya que no estarían completos por sí solos. Las reglas de composición, también se relacionan con todos los requerimientos involucrados en ellas. De esta manera, los intereses y las reglas que se reutilizan tienen vínculos de dependencia con los requerimientos que involucran.

La reutilización de requerimientos (Figura 9.1) se debe llevar a cabo cuando se ha identificado en un proyecto que un requerimiento tiene aplicación variada, que puede incorporarse en otros proyectos distintos, y que su especificación es modular, clara y completa. Se debe evaluar si la especificación del requerimiento ha capturado adecuadamente, de la manera más aislada posible, una característica del sistema que tiene una potencial posibilidad de aparecer de nuevo en otros sistemas.

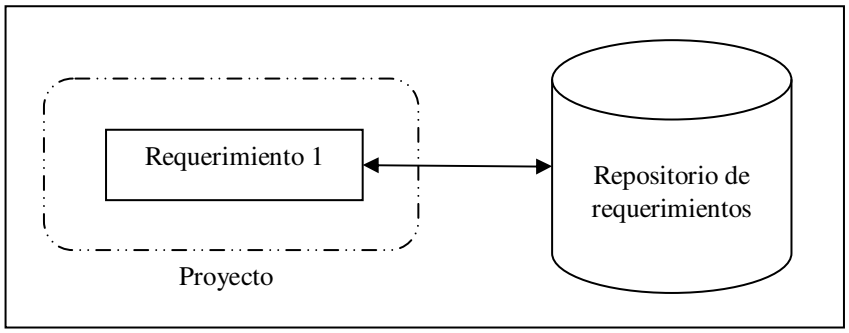


Figura 9.1 Clasificación de un requerimiento.

Se debe considerar reutilizar un interés cuando se identifica que un grupo de requerimientos ha sido encapsulado bajo un solo concepto y esa agrupación ha resultado modular, bien estructurada, completa, y precisa; además que, por el concepto que representa, es viable su incorporación en diferentes proyectos. Al reutilizar un interés se debe tener en cuenta que éste se almacena en conjunto con los requerimientos que contiene (Figura 9.2), por lo que no solo es importante revisar la coherencia del interés sino la integridad individual de sus requerimientos contenidos.

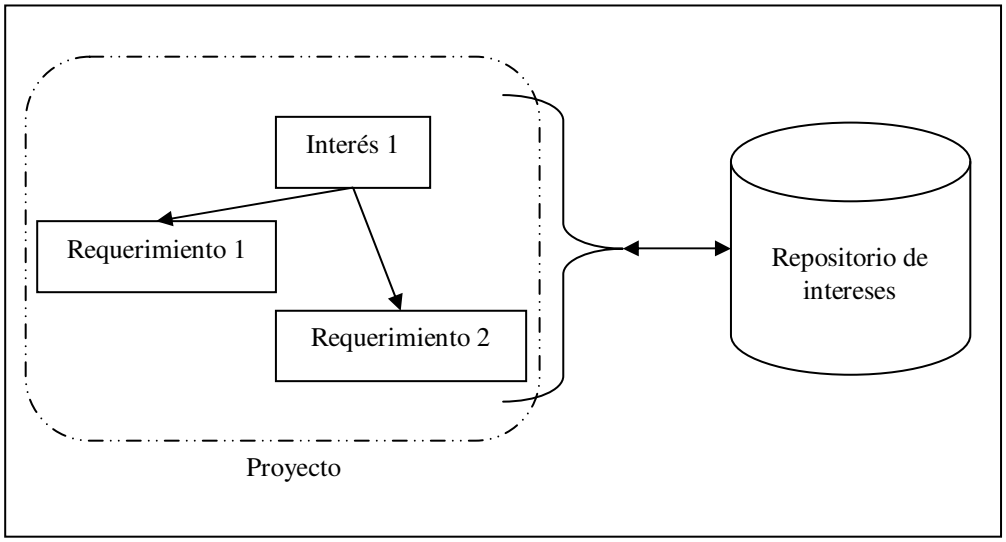


Figura 9.2 Clasificación de un interés.

Por otra parte, la reutilización de reglas de composición involucra el enlace de por lo menos dos requerimientos separados. Si el efecto de ese enlace resulta en una acción clara, precisa, repetible, conceptual y coherente, esa regla es candidata ideal para ser reutilizada. Las reglas son tratadas en conjunto con sus parámetros de configuración y los requerimientos que involucran (Figura 9.3). Si una regla hace referencia a los requerimientos dentro un interés, éste último no se almacena en conjunto con la regla; solo se unifican con ella los requerimientos necesarios para implementar la composición expresada por la regla. Es por eso que para las reglas, también es válida la recomendación de cuidar el contenido de sus requerimientos relacionados.

Quizá las reglas de composición sean los elementos con más difícil oportunidad de reutilización porque involucran requerimientos e intereses y expresan relaciones demasiado particulares entre ellos dentro del contexto de un proyecto. Sin embargo, tiene sentido reutilizarlas si se piensa en un escenario de desarrollo de múltiples proyectos que comparten un mismo dominio, por ejemplo, en una línea de productos, donde los efectos de relación entre los mismos requerimientos pueden presentarse recurrentemente.

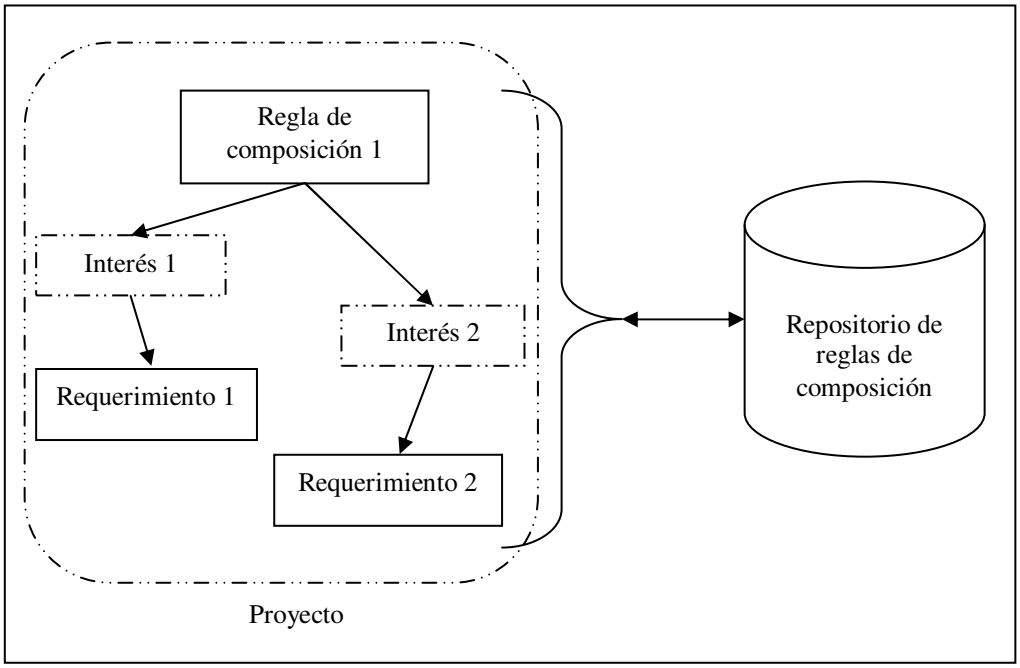


Figura 9.3 Clasificación de una regla de composición.

9.2.2. Instanciación

La reutilización vista desde un proyecto que incorpora elementos existentes en los repositorios conlleva al concepto de instanciación. Mediante la clasificación, los repositorios reciben elementos reutilizables provenientes de diferentes proyectos y los almacenan; mediante la búsqueda e instanciación, esos elementos pueden ser localizados, copiados y adaptados en otros proyectos.

En un proyecto se pueden instanciar requerimientos, intereses o reglas de composición. La instanciación de cualquiera de estos elementos consiste en elegir un elemento del repositorio, adecuarlo y copiarlo al proyecto en el que se trabaja en ese momento. Los cambios que se le realicen dentro del proyecto, se hacen solo dentro de ese contexto y no afectan al elemento que se encuentra en el repositorio. Es decir, el proyecto trabaja siempre con una copia agregada y deja intacta la versión del repositorio.

El proceso de búsqueda es previo a la instanciación y es fundamental para poder ubicar los elementos que pueden ser útiles en escenarios particulares. La búsqueda toma criterios que el especificador construye en base a lo que desea encontrar. El algoritmo contrasta los criterios y el contenido actual de los repositorios para sugerir un conjunto de resultados que cumplen con lo que el especificador indicó. Este es un proceso interactivo, porque una vez que se han encontrado varios elementos, el especificador debe decidir cuál de ellos quiere incorporar al proyecto actual.

La adaptación es un momento de la instanciación donde se configuran los identificadores de los elementos que serán incorporados; su objetivo es evitar conflictos de nomenclatura con los elementos que ya existen en un proyecto. Este proceso también es interactivo: requiere la intervención del especificador para determinar los identificadores que serán asignados. La tarea del algoritmo posteriormente, es verificar que los nuevos nombres elegidos no introduzcan conflictos en el proyecto.

Una vez que se han adaptado los elementos, éstos pasan a ser parte del proyecto actual, como si el especificador los hubiera introducido manualmente desde el principio. Esto significa que luego pueden ser modificados, o incluso eliminados, siempre y cuando estas acciones no comprometan la integridad del proyecto completo.

9.2.3. Reflexión sobre la clasificación y búsqueda de elementos

El modelo base compartido por requerimientos, intereses y reglas que fue presentado en el apartado 7.2, incluye categorías y palabras clave. El algoritmo de reutilización valida que todo elemento que se desee agregar al repositorio tenga por lo menos una palabra en estos campos. Esto garantiza que los elementos sean visibles para el algoritmo de búsqueda cuando se intenten localizar elementos utilizando categorías o palabras clave.

El modelo de acceso a los repositorios propuesto en este trabajo de tesis es básico: considera solo la funcionalidad esencial para almacenar, localizar y extraer elementos de ellos, basándose en criterios de búsqueda que pueden incluir: el identificador del elemento, su descripción, su tipo, o bien, categorías y palabras clave. Sin embargo, es importante reflexionar que los algoritmos que proporcionan el acceso a los repositorios pueden ser cambiados por otros que puedan brindar mayor flexibilidad y eficiencia en las opciones de clasificación, búsqueda y recuperación de elementos. Éstas y otras oportunidades de mejora se exponen en la sección 13.2.

9.3. Algoritmo de reutilización

9.3.1. Objetivo del algoritmo

El algoritmo de reutilización proporciona acceso a los repositorios de la herramienta de software, y actúa en dos puntos diferentes del proceso de desarrollo software: cuando el desarrollador realiza una búsqueda (instanciación), es el encargado de mostrar e incorporar al proyecto actual los elementos encontrados; por otra parte, cuando el desarrollador decide agregar algún elemento del proyecto actual a los repositorios (clasificación), el algoritmo lo toma y lo almacena debidamente categorizado.

Los repositorios contienen elementos que pueden ser incluidos en distintos proyectos. Cuando un elemento se envía a un repositorio, se dice que ha sido clasificado. El proceso inverso a la clasificación es la instanciación. Cuando un elemento del repositorio es seleccionado para incluirse en un proyecto particular, se dice que ha sido instanciado o incorporado. La Figura 9.4 muestra gráficamente ambas acciones.

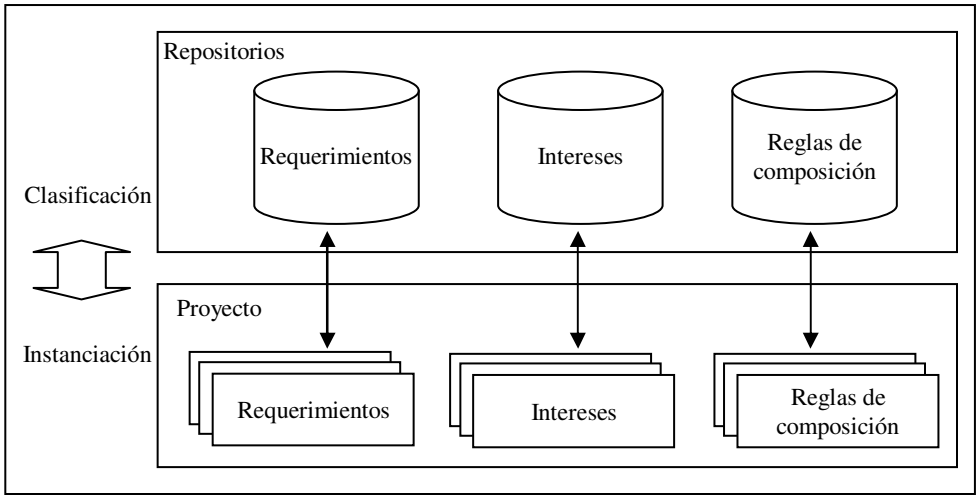


Figura 9.4. Clasificación e instanciación

9.3.2. Diagramas de alto nivel

El proceso de reutilización implementado involucra a dos algoritmos: la instanciación de los elementos (Figura 9.5) y su clasificación (Figura 9.6). El primero comprende la búsqueda de elementos en el repositorio y su incorporación en el proyecto actual, mientras que el segundo se refiere al almacenamiento de nuevos elementos en el repositorio.

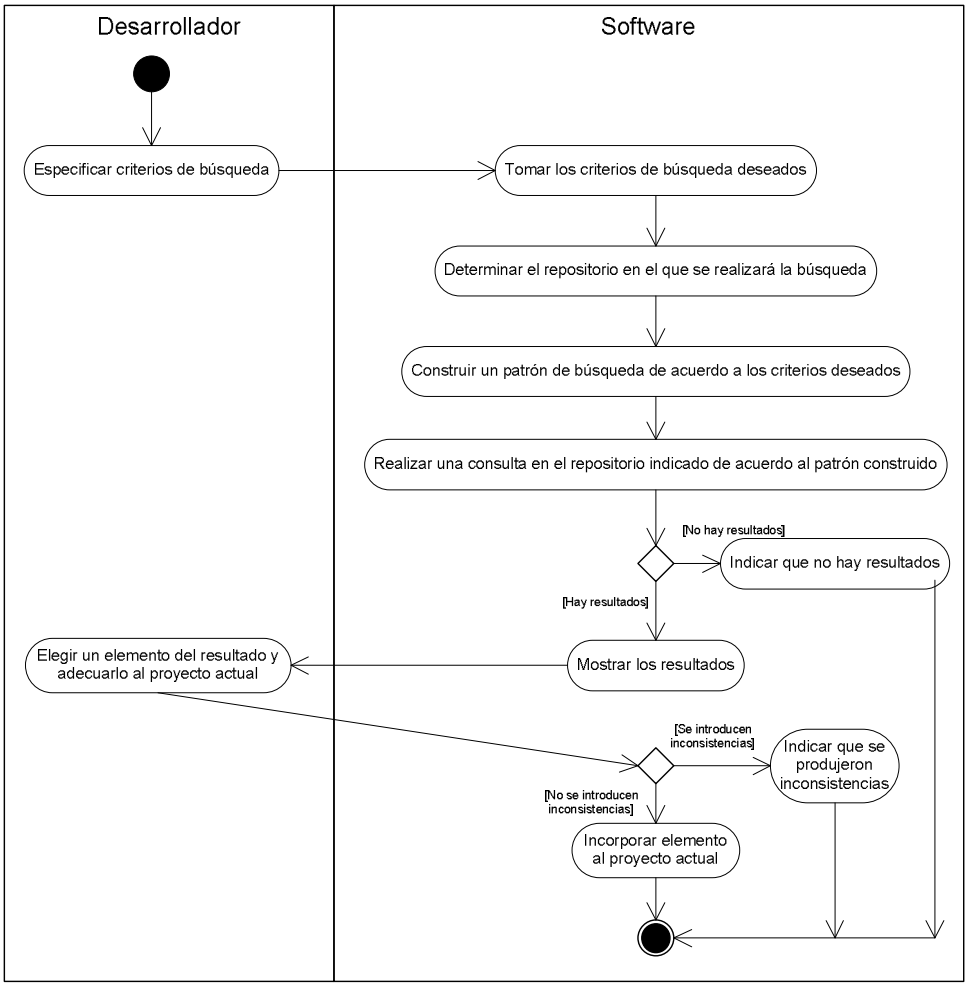


Figura 9.5 Diagrama de alto nivel para el algoritmo de reutilización (fase de instanciación).

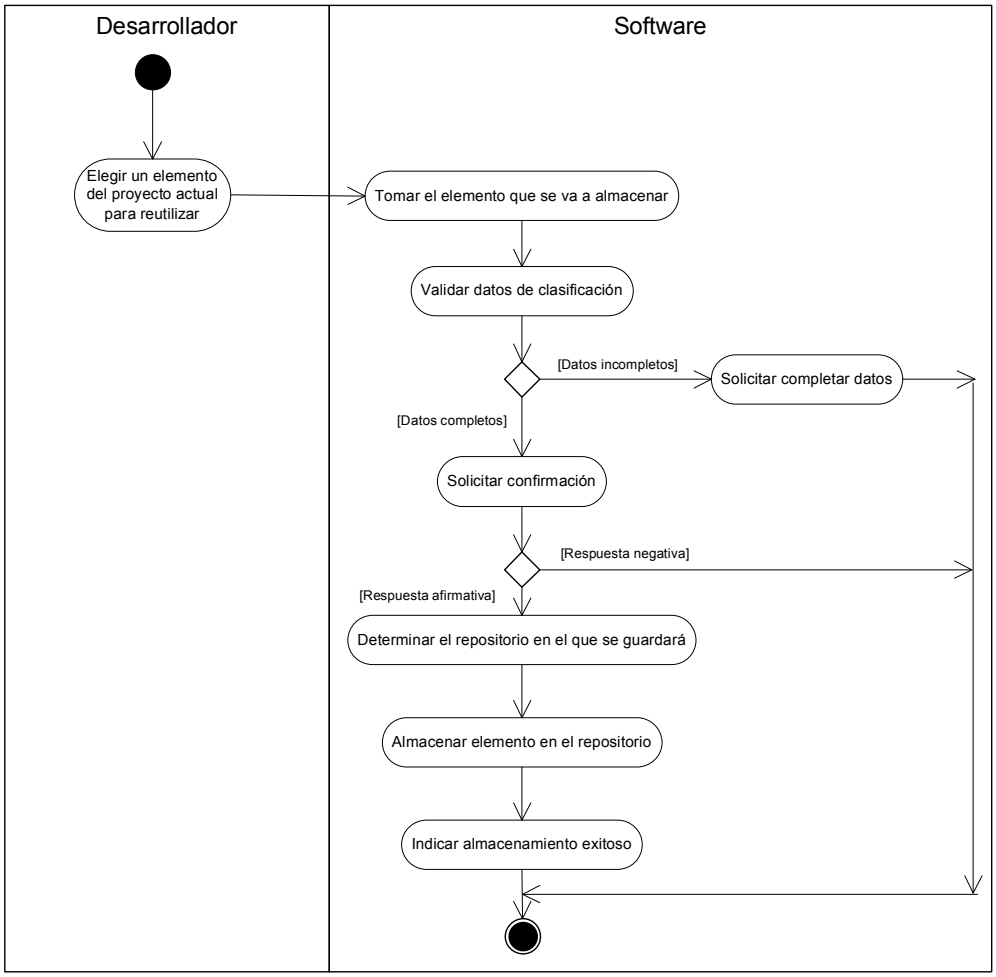


Figura 9.6 Diagrama de alto nivel para el algoritmo de reutilización (fase de clasificación).

9.3.3. Pseudocódigo

En la Figura 9.7 y en la Figura 9.8 se muestran los pseudocódigos más detallados de los dos algoritmos involucrados en el proceso de reutilización que se presentaron en los diagramas de la Figura 9.5 y de la Figura 9.6, respectivamente. La explicación de cada uno se encuentra al final del cuerpo de cada pseudocódigo. Por otra parte, en la Figura 9.9 se listan las subrutinas adicionales que se utilizaron en los algoritmos.

Fase de instanciación

```
Nombre de la rutina:  BUSQUEDA_E_INSTANCIACION_DE_ELEMENTOS
Descripción:          Algoritmo para buscar elementos reutilizables en el repositorio.
Entradas:             Texto que describe los elementos a buscar (Criterio)
                      Campo en el cual buscar (Campo)
                      Tipo de elementos buscados (Tipo[])
                      Proyecto actual (EsteProyecto)
Salidas:              Resultado de la operación (SÍ=Éxito o NO=Fracaso).
Inicio-Rutina
    /* Declaraciones: */
    1)  BuscarEn : Repositorio
    2)  Prototipo, Selección : Elemento

    /* Inicia el algoritmo: */
    3)  Elementos = Vacío
    4)  Leer Criterio, Campo, Tipo[]
    5)  Para Cada t en Tipo
    6)  En Caso (t) Sea
                a.  Requerimiento:
                          BuscarEn = Repositorio de Requerimientos
                          Prototipo = Nuevo Requerimiento con los criterios de búsqueda en
                          los campos indicados.
                b.  Interés:
                          BuscarEn = Repositorio de Intereses
                          Prototipo = Nuevo Interés con los criterios de búsqueda en los
                          campos indicados.
                c.  Regla:
                          BuscarEn = Repositorio de Reglas
                          Prototipo = Nueva Regla con los criterios de búsqueda en los
                          campos indicados.
    7)  Fin-En-Caso
    8)  Agregar a Elementos los resultados de:  Consultar (BuscarEn, Prototipo)
    9)  Fin-Para-Cada
    10) Si Elementos no está vacío
            a.  Sugerir (Elementos)
        Si-No
            b.  Regresar NO. (No hay resultados para esa búsqueda).
        Fin-Si
    11) Leer Selección
    12) Si Selección no está vacío
            a.  Si Agregar(Selección,EsteProyecto)=Verdadero
                     i.  Regresar SÍ.  (Instanciación exitosa)
                Si-No
                    ii.  Regresar NO. (Se introducen inconsistencias)
                Fin-Si
        Si no
            b.  Regresar NO. (La operación fué cancelada)
        Fin-Si
Fin-Rutina
```

Figura 9.7 Pseudocódigo para la fase de búsqueda e instanciación.

- Pasos 1 y 2: se definen los tipos utilizados.
- Paso 3: antes de iniciar con el proceso de búsqueda, se inicializa la estructura *Elementos* con un valor vacío; es ahí donde posteriormente se va a almacenar el resultado encontrado.
- Paso 4: se deben leer los datos de entrada, los cuales definirán los elementos que serán incluidos en el conjunto de resultados de la búsqueda.

- Paso 5: se realiza una iteración para cada tipo de elemento que se desee buscar en el repositorio. Se pueden buscar Requerimientos, Intereses o Reglas de Composición, o cualquier combinación de estos elementos al mismo tiempo.

- Paso 6: se determina el repositorio en el que se llevará a cabo la búsqueda. Dependiendo de los criterios indicados, también se construirá un elemento prototipo que contenga un patrón que servirá para encontrar los elementos que coincidan con él.

- Paso 7: fin del bloque de decisión del paso 6.

- Paso 8: una vez que se conoce el repositorio donde se llevará a cabo la búsqueda y se tiene el elemento prototipo, en este paso se realiza la búsqueda correspondiente y se van acumulando los resultados encontrados.

- Paso 9: fin de la iteración del paso 5.

- Paso 10: se verifica que existan resultados como producto de la búsqueda realizada. Si existen, se dan a conocer; si no, el proceso termina indicando que no se produjeron resultados.

- Paso 11: el desarrollador tiene la oportunidad de elegir uno de los elementos mostrados para incorporarlo al proyecto.

- Paso 12: se verifica si eligió un elemento, en cuyo caso, el paso 12-a intenta incorporarlo al proyecto actual. Si la instanciación no produce inconsistencias, se indica que el proceso ha tenido éxito; pero si se detectan inconsistencias, se indica que la instanciación no se ha podido llevar a cabo. El paso 12-b indica que a pesar de haber obtenido resultados de la búsqueda, la operación de instanciación fue cancelada por el desarrollador.

Fase de clasificación

```
Nombre de la rutina:  ALMACENAMIENTO_ELEMENTOS
Descripción:          Algoritmo para almacenar elementos reutilizables en el
                      repositorio.
Entradas:             Elemento a almacenar (Elem)
Salidas:              Respuesta positiva o negativa para saber si el elemento se pudo
                      almacenar
Inicio-Rutina
    /* Declaraciones: */
    1)  Rep : Repositorio

    /* Inicia el algoritmo: */
    2)  Si Elem tiene datos de clasificación asignados
            a)  Si Confirmar() Entonces
                      i.    En Caso (Elem) Sea
                                    •   Requerimiento:
                                            Rep = Repositorio de Requerimientos
                                    •   Interés:
                                            Rep = Repositorio de Intereses
                                    •   Regla:
                                            Rep = Repositorio de Reglas
                     ii.    Fin-En-Caso

                    iii.    Almacenar(Elem, Rep)
                     iv.    Devolver una respuesta afirmativa como resultado de la
                            ejecución de la rutina.
            b)  Si-No
                        i.  Devolver una respuesta negativa como resultado de la
                            ejecución de la rutina.
            c)  Fin-Si

    3)  Si-No
            a.  Devolver una respuesta negativa como resultado de la ejecución de la
                rutina.
    4)  Fin-Si
Fin-Rutina
```

Figura 9.8 Pseudocódigo para la fase de clasificación.

- Paso 1: se definen los tipos utilizados.

- Paso 2: se verifica si el elemento que se va almacenar tiene asignados los datos de clasificación para poder ubicarlo posteriormente en el repositorio. Si el elemento tiene asociados sus datos de clasificación, se procede con el paso 2a, en donde se identifica el tipo del elemento para determinar el repositorio en el cual se va a almacenar. Una vez que se conoce el repositorio destino, el elemento se agrega en él y se indica que el elemento pasó la etapa de validación, por lo que el proceso fue exitoso.

- Paso 3: si la condición del paso 2 no fue verdadera, significa que el elemento que se desea almacenar no tiene asociados los datos de clasificación, por lo que en el paso 3 se indica que el elemento no se puede almacenar en el repositorio por estar incompleto.

- Paso 4: fin de la condición del paso 2.

```
Nombre de la subrutina: CONSULTAR
Descripción: Lleva a cabo la consulta de elementos en el repositorio
Datos de entrada: Repositorio en el que se hará la búsqueda,
                  Prototipo del elemento deseado
Datos de salida: Un conjunto de elementos que existen en el repositorio y que coinciden con
                 el criterio de búsqueda descrito por el elemento prototipo.
Inicio-Subrutina
        Utilizando el prototipo, realizar la búsqueda de elementos coincidentes en el
        repositorio y regresar todos los elementos encontrados.
Fin-Subrutina


Nombre de la subrutina: SUGERIR
Descripción: Da a conocer un conjunto de elementos que son producto de una consulta previa.
Datos de entrada: Conjunto de Elementos para mostrar
Datos de salida: Ninguno
Inicio-Subrutina
        Dar a conocer cada uno de los elementos que se reciben como entrada.
Fin-Subrutina


Nombre de la subrutina: ALMACENAR
Descripción: Almacena un elemento en el repositorio
Datos de entrada: Elemento a almacenar
                  Repositorio en el que se va a guardar el elemento
Datos de salida: Ninguno
Inicio-Subrutina
        Guardar el elemento en el repositorio indicado
Fin-Subrutina


Nombre de la subrutina: CONFIRMAR
Descripción: Preguntar al desarrollador si está seguro de reutilizar un elemento.
Datos de entrada: Ninguno
Datos de salida: Valor Verdadero o Falso
Inicio-Subrutina
        Preguntar al desarrollador si está seguro de reutilizar el elemento y devolver su
        respuesta como resultado de esta subrutina.
Fin-Subrutina


Nombre de la subrutina: AGREGAR
Descripción: Incorporar un elemento a un proyecto.
Datos de entrada: Elemento a integrar al proyecto
                  Proyecto al cual se va a integrar
Datos de salida: Valor Verdadero o Falso
Inicio-Subrutina
        Agrega un elemento a un proyecto. Devuelve Verdadero si el proceso fue exitoso, o
        Falso en caso contrario.
Fin-Subrutina

Nombre de la subrutina: HAYINCONSISTENCIAS
Descripción: Verifica si hay inconsistencias en el proyecto
Datos de entrada: Proyecto que se va a revisar
Datos de salida: Valor Verdadero o Falso
Inicio-Subrutina
        Verifica si hay inconsistencias en un proyecto. Devuelve verdadero si encuentra
alguna, o Falso en caso contrario.
Fin-Subrutina
```

Figura 9.9 Principales subrutinas utilizadas en los pseudocódigos de reutilización.

9.4. Conclusiones

La reutilización de especificaciones es factible entre diferentes proyectos mediante el uso de repositorios comunes que almacenan requerimientos, intereses y reglas de composición. El uso repetido de estos elementos permite aprovechar la experiencia de trabajos anteriores; esto se lleva a cabo en dos puntos del proceso de desarrollo: el de clasificación y el de

instanciación. La clasificación y la instanciación permiten, respectivamente, almacenar elementos en el repositorio e incorporar elementos de un proyecto en otros diferentes. Ambos procedimientos se refieren a la utilización repetida de especificaciones pero desde diferentes perspectivas. Ambas tareas discurren en momentos diferentes y requieren de la interacción del especificador.

La reutilización de especificaciones en general es un proceso que debe realizarse cautelosamente. A pesar de que cualquier requerimiento, interés o regla de composición puede ser técnicamente reutilizado, se recomienda revisar sus características para mantener unos buenos parámetros de calidad en los elementos de los repositorios y, evitar de esta manera, la propagación de elementos defectuosos o incompletos en otros proyectos.

Capítulo 10.
El prototipo de software

10.1. Introducción

El apoyo de una herramienta de software que permita automatizar los procesos de especificación, composición y reutilización, es primordial en Z-SIIMD. Para el presente trabajo se desarrolló un prototipo nombrado SPZS (Software Prototipo para Z-SIIMD), cuya principal meta es facilitar una implementación del modelo de trabajo propuesto por Z-SIIMD. Los principales objetivos identificados que marcaron la línea de desarrollo del prototipo son:

a) Gestionar los proyectos.
b) Gestionar los repositorios.
c) Proporcionar medios para introducir requerimientos, intereses y reglas de composición.
d) Implementar y validar el algoritmo de composición del modelo.
e) Presentar el modelo previo y posterior a la composición.
f) Cuidar la integridad del modelo ante los cambios.

10.2. Análisis y diseño del prototipo

Las funcionalidades de SPZS se muestran en la Figura 10.1. Con ayuda de este software (ver Figura 10.2), el usuario mantiene un modelo de especificaciones dentro de un proyecto, el cual está formado por tres tipos básicos de elementos: requerimientos, intereses y reglas de composición. Estos elementos pueden ser incorporados al proyecto directamente por el usuario a partir de cero, o bien, mediante instanciación, para lo cual, debieron existir previamente en el repositorio.

Los repositorios se presentan como un solo contenedor global para todos los proyectos, el cual mantiene de forma ordenada los intereses, los requerimientos y las reglas de composición. En el repositorio se pueden almacenar elementos nuevos (Figura 10.3); este proceso ocurre cuando el usuario selecciona estratégicamente un elemento del proyecto actual y lo almacena para que sea reutilizado en uno o más proyectos posteriores.

Cuando se toman elementos del repositorio para utilizarlos en el proyecto, se dice que éstos han sido instanciados. La instanciación es un proceso que involucra la búsqueda de elementos, la selección y la copia de los mismos en el proyecto actual (Figura 10.4). Una vez que se ha llevado a cabo la instanciación, los cambios realizados a estos elementos son exclusivos para el proyecto en el que se encuentran. Esto quiere decir que la estructura

original del repositorio se mantiene intacta, y las adecuaciones que el usuario realice, se manifiestan únicamente en el proyecto actual.

SPZS mantiene identificadores únicos dentro de cada proyecto, por lo que al momento de instanciar un elemento se debe revisar el nombre que tiene, y si es necesario, debe cambiarse para que no ocurran conflictos de nomenclatura.

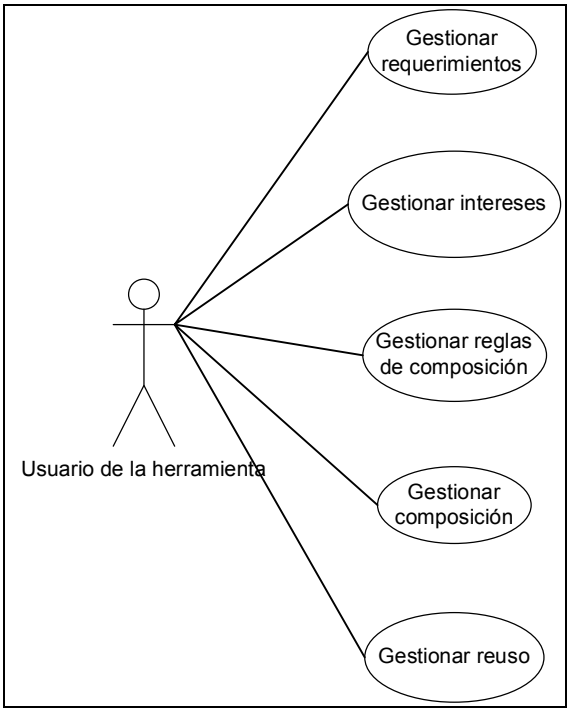


Figura 10.1. Casos de uso del prototipo de software SPZS.

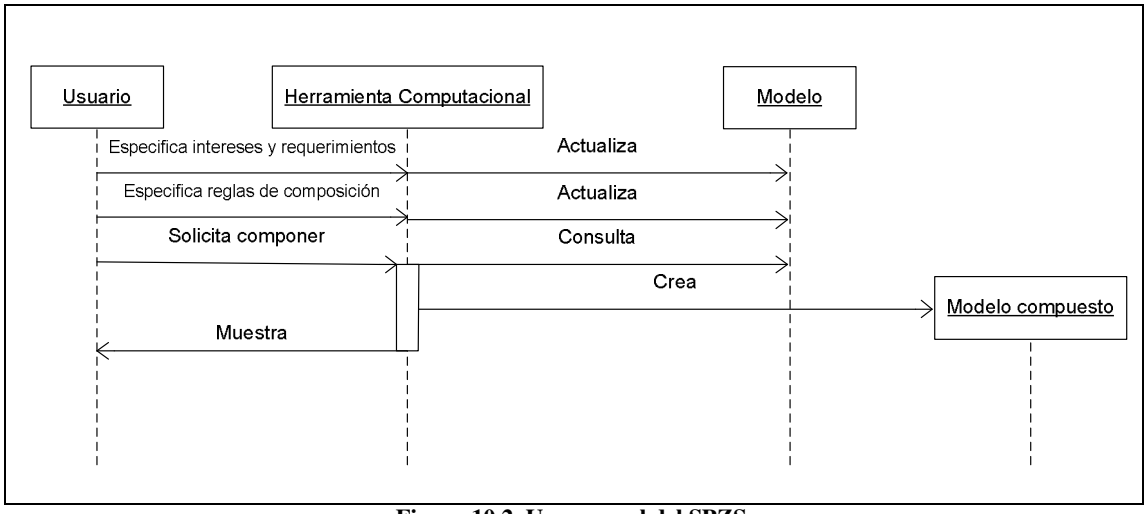


Figura 10.2. Uso general del SPZS.

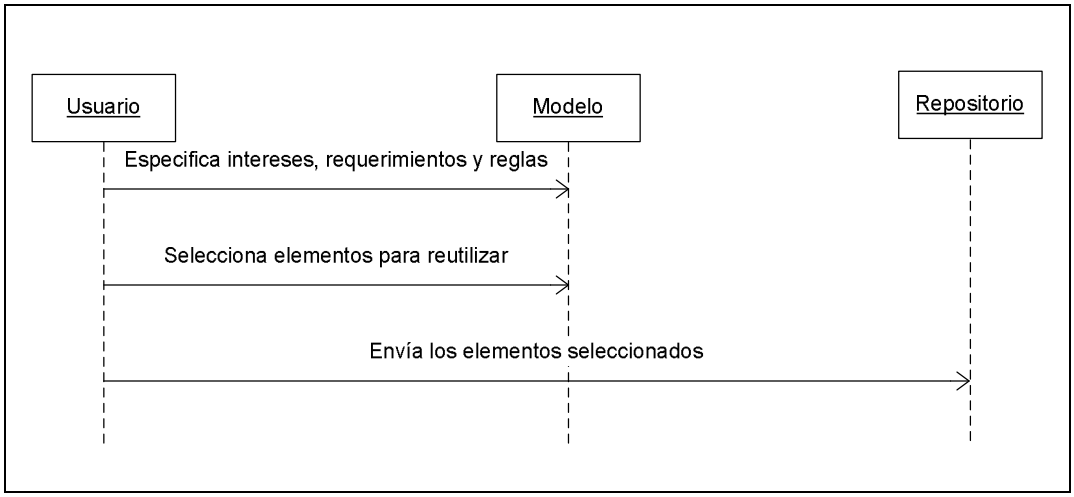


Figura 10.3. Proceso de reutilización de elementos.

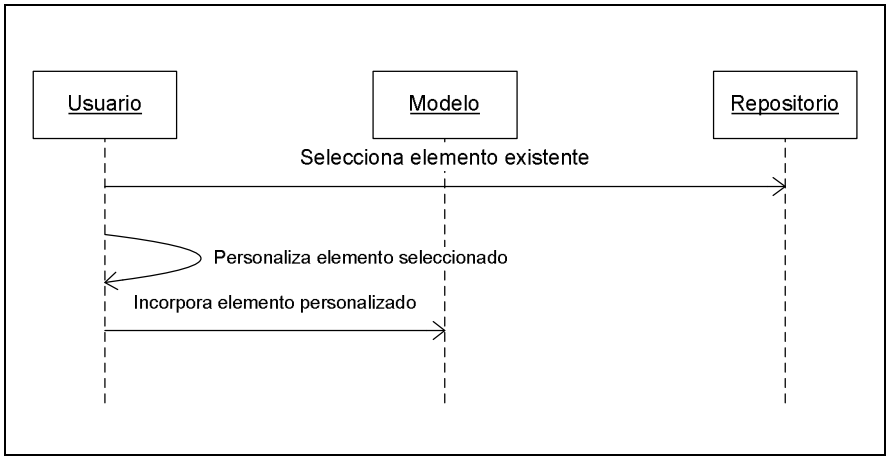


Figura 10.4. Proceso de instanciación de elementos.

10.3. Funcionalidad del prototipo

La programación de SPZS se llevó a cabo en una aplicación para Windows, utilizando *C#* (NAGEL, Christian et al., 2008) con el entorno de *Visual Studio 2008 Express Edition*. Para la gestión de almacenamiento se utilizó la base de datos orientada a objetos *db4objects*.(EDLICH, Stefan et al., 2006)

El software posee un menú cuyas principales opciones sirven para gestionar los proyectos realizados por el usuario con operaciones como creación, almacenaje, y apertura de archivos de especificaciones.

La interfaz gráfica también incluye una serie de paneles con pestañas, donde se puede trabajar con los intereses, requerimientos y reglas de composición, así como también con el modelo completo y el repositorio.

En toda la aplicación se usan uniformemente los botones para agregar, eliminar, editar y reutilizar elementos en diferentes escenarios según corresponda. Otra característica común es la posibilidad de dar *doble click* sobre cualquier elemento que se encuentre en una cuadrícula o *grid*, para desplegar una ventana emergente con los datos asociados a él.

En la primera pestaña se muestra un listado con los intereses que han sido capturados en el sistema (ver Figura 10.5). Cuando se editan los datos de un interés (ver Figura 10.6), se tiene acceso a textos para especificar su identificador, descripción, categorías a las que pertenece, y palabras clave que ayuden a ubicarlo si llega a estar en el repositorio; de manera informativa se muestra también la evolución que ha tenido ese interés desde que fue creado en el proyecto. Desde esta pestaña también se tiene acceso a los requerimientos asociados con ese interés.

Cada requerimiento también tiene un identificador, una descripción, datos informativos de evolución, categorías y palabras clave que ayudan a ubicarlo. Además incorpora un esquema para introducir su especificación formal en dos secciones: declaraciones e invariantes (ver Figura 10.7), que corresponden a un esquema tradicional de Lenguaje Z. En cada renglón de la especificación formal existe la posibilidad de agregar comentarios en lenguaje natural que expliquen con detalle la intención de esa parte de la especificación.

En la pestaña de las reglas de composición existen dos sub pestañas: una es para mostrar el resumen general de las reglas mostrando solo identificadores y descripciones (ver Figura 10.8); y la segunda es para capturar el detalle completo de cada una (ver Figura 10.9) donde se incluye: el identificador y descripción para la regla, así como también los datos de los dos operandos involucrados y sus respectivas versiones a las cuales se desea hacer referencia. También se capturan los valores para las cláusulas: `@WHERE`, `@NEWNAME`, `@CONN`, `@ADDVAR`, `@CHG`, `@REP`, y `@COND`.

Al momento de introducir las reglas de composición se cuenta con la facilidad de consultar los datos completos de sus intereses o requerimientos asociados dando *click* con el *ratón* sobre el signo de interrogación marcado como hipervínculo.

Desde la pestaña del modelo completo (ver Figura 10.10), se puede seleccionar la generación del mismo y el filtrado de los requerimientos para mostrar; las siguientes opciones están disponibles: desplegar todos los requerimientos, solo los compuestos, solo los originalmente especificados, o solo los nuevos que fueron creados con un nuevo nombre. También es posible exportar los requerimientos que se encuentran visibles en la cuadrícula en ese momento hacia un archivo de texto plano.

Desde la pestaña del repositorio (ver Figura 10.11), se gestionan los elementos que ya se encuentran almacenados y listos para reutilizarse. Se pueden obtener listados completos o parciales filtrados por identificador, descripción, categoría o palabra clave. Existe la opción de concentrar la búsqueda solo en intereses, requerimientos, o reglas de composición, o cualquier combinación de estas opciones. También desde esta sección se puede realizar el borrado y la instanciación de elementos del repositorio.

Al momento de instanciar un elemento en el proyecto actual, el software muestra una ventana emergente donde el especificador puede revisar y modificar los identificadores que se usarán en el proyecto con la finalidad de evitar colisiones con los nombres de otros elementos existentes. Por ejemplo, la Figura 10.12 muestra la pantalla emergente que se obtiene al instanciar una regla de composición; ahí el especificador indica los nuevos identificadores para la regla, los intereses y requerimientos asociados. El software sugiere los mismos nombres que tienen los elementos en el repositorio; éstos pueden ser utilizados tal como están, siempre y cuando no causen conflictos (SPZS indica si ocurre esta condición), o bien, pueden ser cambiados según las necesidades de convención de nombres en el proyecto actual.

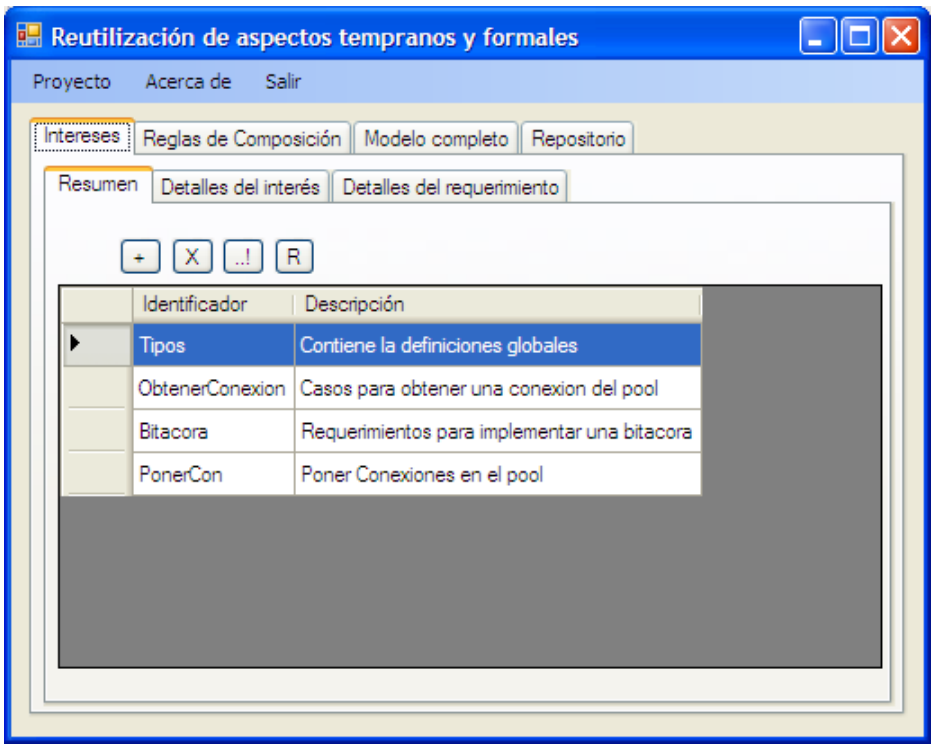


Figura 10.5. Sección de resumen de intereses.

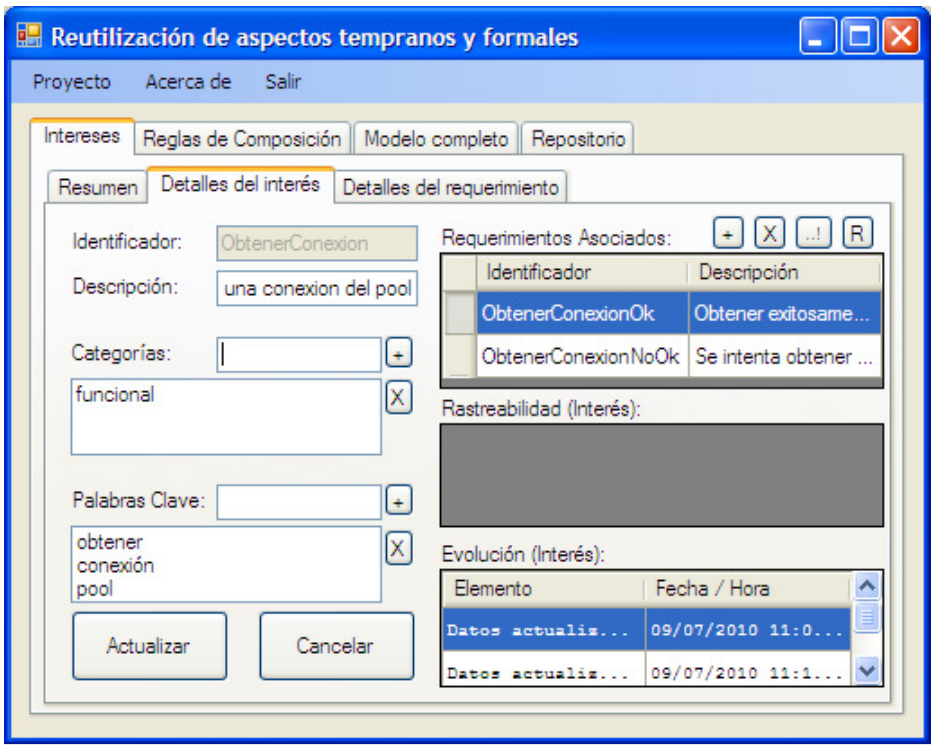


Figura 10.6. Sección de detalles de un interés.

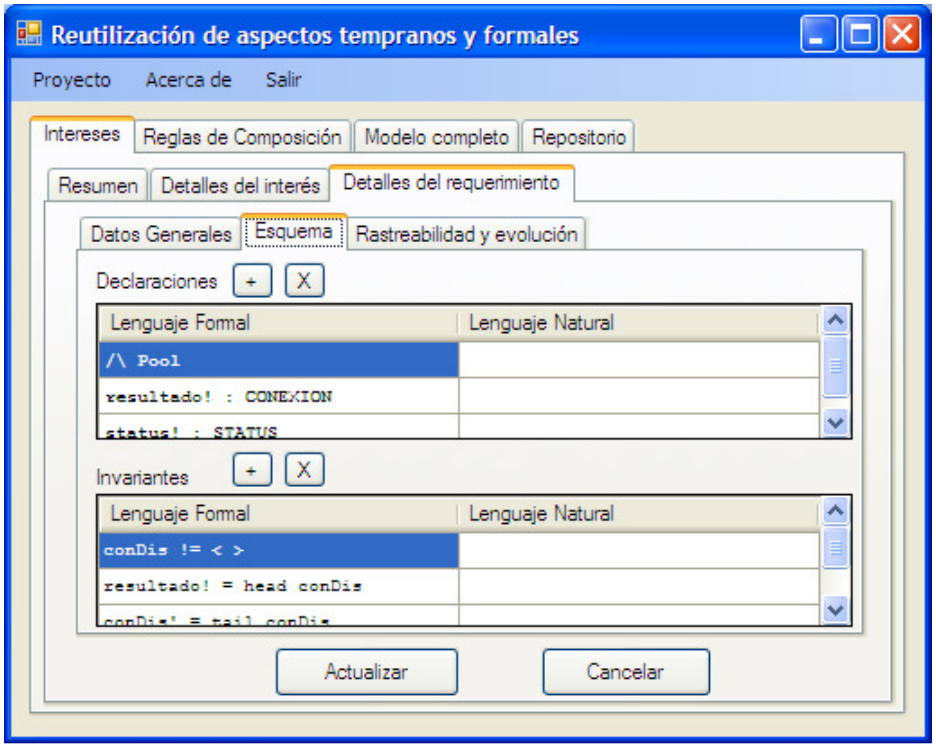


Figura 10.7. Esquema formal de un requerimiento.

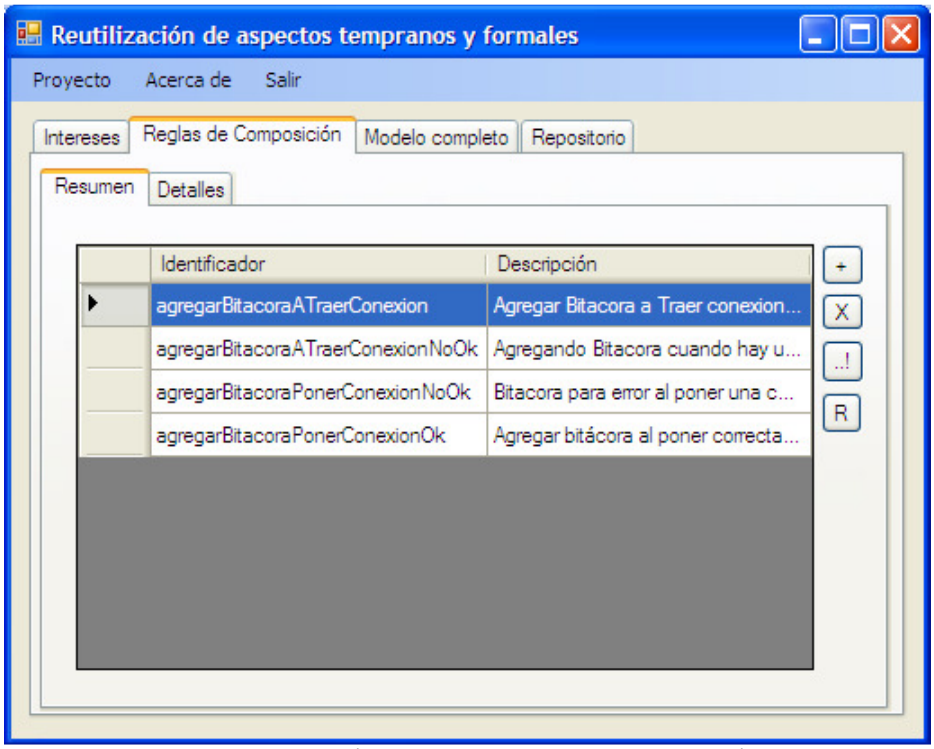


Figura 10.8. Sección de resumen de reglas de composición.

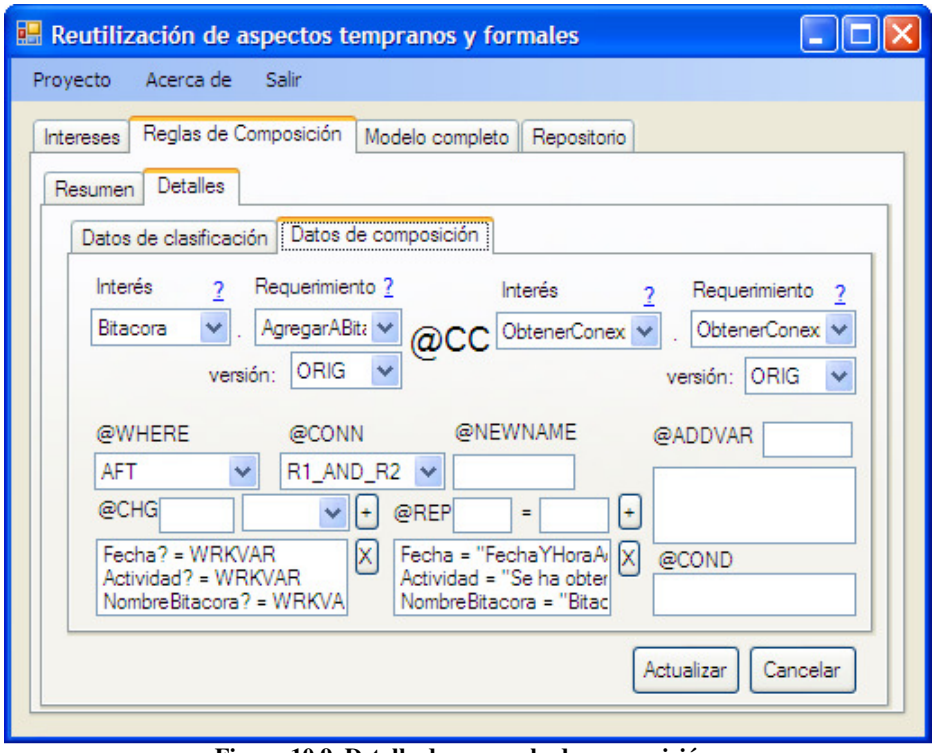


Figura 10.9. Detalle de una regla de composición.

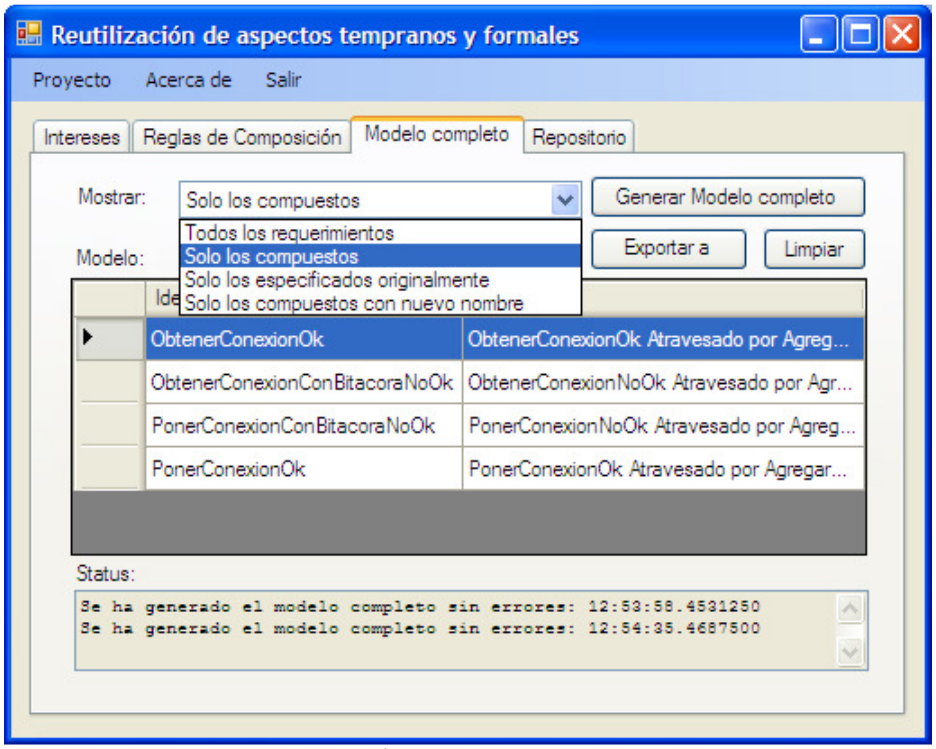


Figura 10.10. Sección para obtener el modelo completo.

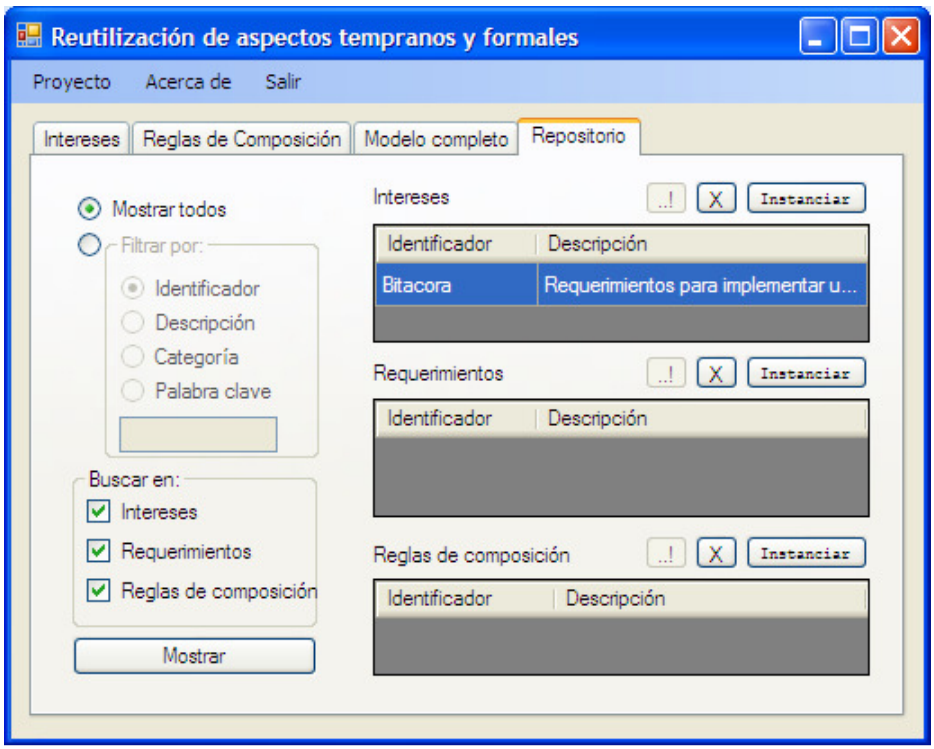


Figura 10.11. Repositorio

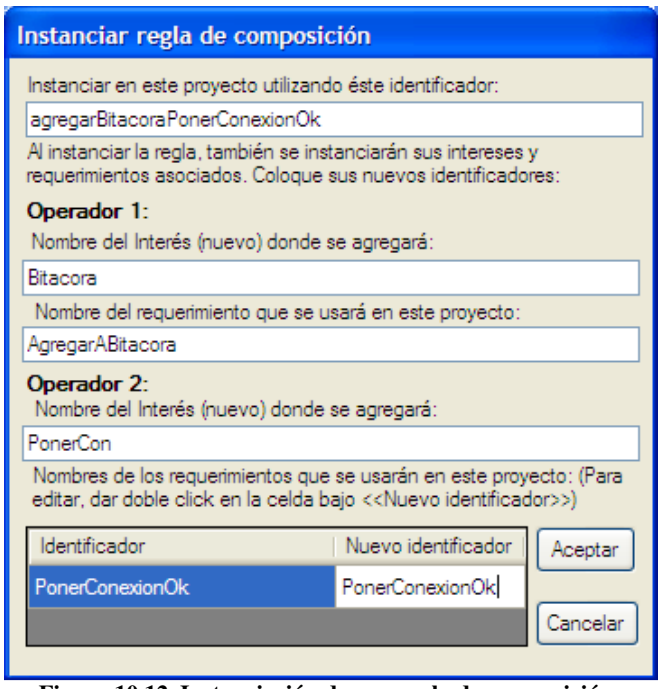


Figura 10.12. Instanciación de una regla de composición.

10.4. Alcance del prototipo

El software SPZS fue realizado como un medio de soporte para esta tesis e incluye características básicas plenamente funcionales y suficientemente representativas. Durante el desarrollo de este software, se hicieron valoraciones para encaminar el esfuerzo hacia las características más relevantes, dejando otras para implementarse en versiones posteriores.

A continuación se muestran algunas restricciones de la funcionalidad ofrecida por el primer prototipo SPZS:

1) Implementación limitada del modelo multidimensional de separación de intereses.- Siguiendo los principios del modelo multidimensional, la versión actual de SPZS permite especificar separadamente cada uno de los intereses relacionados, sin imponer descomposición primaria alguna; también permite la creación de reglas separadas que establecen la forma de impacto de los intereses entre sí, de esta manera se pueden manejar los intereses transversales adicionalmente a los no transversales; sin embargo aun falta por implementar el manejo de diferentes dimensiones de interés simultáneas. Si bien actualmente los intereses no cambian su estructura al modificar la descomposición inicial, obtener una nueva descomposición diferente es posible pero no es tarea directa; para lograrlo, se deben realizar los respectivos cambios en las reglas de composición y en la selección de intereses relacionados, pasando también por un proceso de exportación a texto plano.

SPZS tampoco realiza un mapeo directo de terminología del modelo multidimensional genérico; no incorpora términos como hiper sección, o hiper módulo, aunque implícitamente implementa versiones limitadas de estos conceptos. Sin embargo, SPZS mantiene la idea central del modelo en cuanto a la separación independiente de intereses; y se contempla que en la siguiente versión se incorporará el manejo de dimensiones, implementando el mismo algoritmo de composición y utilizando las mismas reglas, pero permitiendo un manejo más flexible de las descomposiciones en el proyecto. En los primeros trabajos de esta siguiente versión se está estudiando la nueva estructura de las opciones. Una aproximación inicial se muestra en la Figura 10.13.

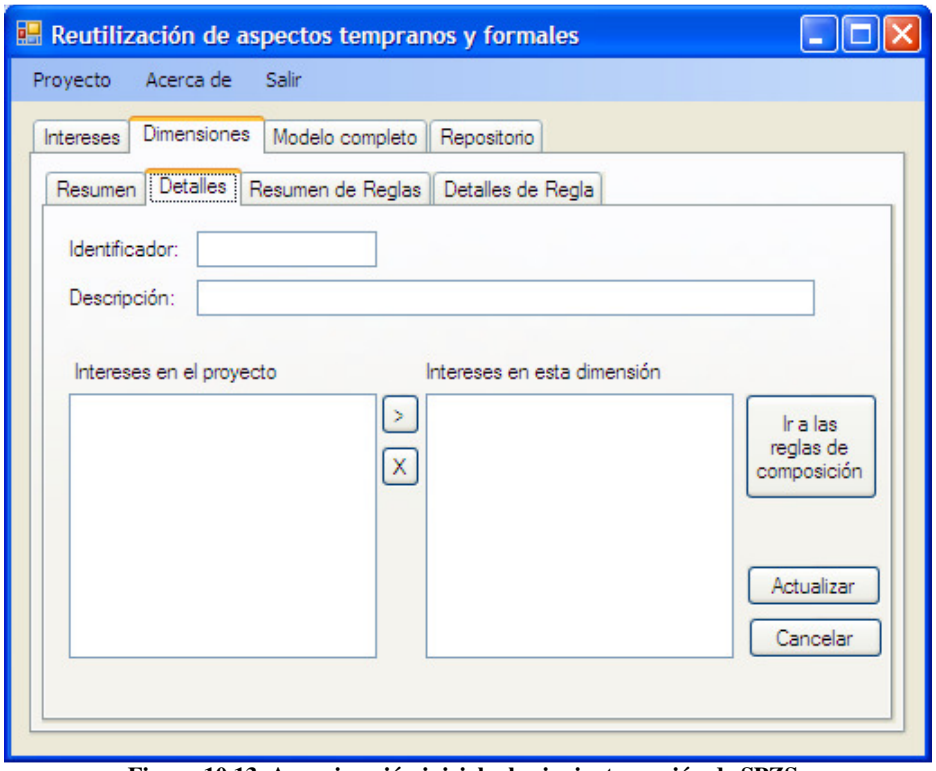


Figura 10.13. Aproximación inicial a la siguiente versión de SPZS.

2) Captura de las especificaciones formales.- Cuando el usuario introduce los requerimientos a partir de cero, SPZS le proporciona medios para capturar las especificaciones formales; sin embargo, éstas se introducen utilizando texto libre y no se restringen a la sintaxis de lenguaje Z.
En las pruebas realizadas, el texto capturado se adaptó para que siguiera la sintaxis estándar de Z lo más cercanamente posible. Se encontró que la simbología meramente gráfica de Z puede ser representada con caracteres mnemónicos de texto plano tomando en cuenta el grado de similitud que tengan uno o más caracteres consecutivos con los símbolos gráficos originales. Por ejemplo: el símbolo Ξ puede ser emulado mediante la tilde encerrada en paréntesis cuadrados [~] y el símbolo Δ mediante las dos barras diagonales /\ .

SPZS asume que la entrada de datos se realiza utilizando un conjunto uniforme de convenciones simbólicas de texto, cuya labor de definición inicial queda a cargo del especificador.

3) Restricciones de organización en los intereses y requerimientos de un proyecto.- A pesar de que en el modelo de Z-SIIMD el uso de intereses es opcional, la interfaz de usuario de la versión actual de SPZS requiere que cada requerimiento capturado pertenezca a un interés.

4) Los elementos compuestos deben almacenarse o instanciarse en conjunto con sus elementos contenidos.- Cuando se trata de un interés, éste se almacena en el repositorio con todos los requerimientos contenidos en él. Cuando se trata de una regla de composición, ésta se almacena con todos los requerimientos involucrados en ella. Al momento de instanciar intereses y reglas, se aplica esta misma restricción; en este caso, SPZS proporciona medios para revisar los identificadores de los elementos y evitar conflictos con los nombres que ya existen en el proyecto.

5) Categorización de los elementos.- Los intereses y requerimientos tienen categorías y palabras clave asociadas a ellos que permiten ubicarlos correctamente en el repositorio. Estos campos son introducidos en texto libre para cada caso.

6) Formatos de presentación de modelo del proyecto.- SPZS presenta en pantalla el modelo de requerimientos original capturado por el usuario y el modelo compuesto total o parcial, producto de la acción de las reglas de composición, y permite su exportación a un archivo solo en formato de texto plano.

7) Mecanismos de trazabilidad de los elementos.- No se incluyó en la versión funcional de SPZS una manera en la que el especificador pueda llevar un registro personal de la trazabilidad de los requerimientos, intereses y reglas a lo largo del ciclo de vida del proyecto. Esta característica resultaría en un mapa de trazabilidad que relacionaría cada elemento de la especificación con las subsecuentes actividades de diseño, implementación, pruebas, etc.

10.5. Conclusiones

El software SPZS fue desarrollado como un prototipo incremental para dar soporte al modelo de trabajo de Z-SIIMD. SPZS implementó funcionalidades que permitieron determinar que la propuesta se encontraba bien encaminada en lo referente a los algoritmos de composición y búsqueda y reutilización. La versión actual más estable de SPZS se describió en este capítulo, y también se abordaron las consideraciones especiales que se tomaron en su desarrollo.

Capítulo 11.
Aplicación del proceso de composición de especificaciones. Caso de uso: agrupamiento de conexiones

11.1. Introducción

En este capítulo se presenta un ejemplo de aplicación donde se utiliza el comportamiento de un agrupamiento de conexiones (*connection pool*) para demostrar, primero, la separación de intereses, luego la especificación de interacciones entre los intereses aislados y finalmente la obtención de un modelo completo guiado por las reglas de composición. Para el ejemplo se hace énfasis en la especificación formal de requerimientos y reglas de composición utilizando Lenguaje Z y Z-SIIMD.

El escenario aquí presentado se retoma posteriormente en el Capítulo 12, donde se capturan estas especificaciones en SPZS para mostrar el proceso de reutilización de un interés en otro proyecto diferente.

El caso de uso de este capítulo propone modelar las principales funcionalidades que ofrece un agrupamiento de conexiones a una base de datos. Un agrupamiento permite reutilizar recursos que son costosos de crear a partir de cero cada vez que se necesitan. El agrupamiento actúa como una entidad recicladora que proporciona medios para colocar un recurso en él, es decir, agregarlo para reutilizarlo posteriormente, o bien para obtener un recurso reciclado, es decir, sacarlo para usarlo en ese momento.

Para este ejemplo, se va a utilizar 100 como el número máximo de conexiones que el agrupamiento puede almacenar; también se va a considerar que el manejo de las conexiones sigue una filosofía FIFO (*First Input, First Output*), así que la conexión más antigua es la que primero se utiliza, el resto se va agregando a una línea de espera y permanece ahí hasta que corresponda su turno al necesitarse más conexiones.

Adicionalmente, se desea dotar de un mecanismo de bitácora a cada una de las funcionalidades del agrupamiento de conexiones. Una bitácora, también llamada *log*, es un registro detallado de actividades. El propósito de utilizar esta técnica es entender la conducta de un sistema (LADDAD, Ramnivas, 2003), si se utiliza como un medio de depuración. También puede servir como un medio de inspección de la funcionalidad que ha sido utilizada en un sistema.

Las entradas de una bitácora típicamente involucran la fecha y la hora de registro, una descripción de la operación realizada, y los datos adicionales que sean necesarios para mostrar detalles más completos de las acciones registradas. En el presente ejemplo se trabajará con una bitácora para asentar una evidencia cada vez que el agrupamiento de conexiones sea utilizado. En la bitácora se registrará para cada operación: una breve descripción de la acción ejecutada, así como la fecha y hora en la que ésta fue realizada.

11.2. Modelado de agrupamiento de conexiones

Los tipos para esta especificación son dos: *CONEXION*, que representa una conexión reutilizable hacia una base de datos; y *STATUS*, que representa el resultado de ejecutar una operación de agregar u obtener una conexión del agrupamiento.

$$[CONEXION]$$
$$STATUS ::= OKConexionObtenida \mid OKConexionAgregada \mid ErrorNoHay \mid ErrorMaxAlcanzado$$

Ahora se modelará el agrupamiento de conexiones mediante el esquema *P*

$$\begin{array}{|l}
\hline
\textit{P} \\
conDis : seq\ CONEXION \\
\hline
\#conDis \leqslant MAXConexiones \\
\hline
\end{array}$$

y se indicará la capacidad máxima del agrupamiento mediante el valor global *MAXConexiones*

$$\begin{array}{|l}
MAXConexiones : \mathbb{Z} \\
\hline
MAXConexiones = 100
\end{array}$$

El objeto *conDis* dentro del esquema *P* representa las conexiones que están listas para ser utilizadas de nuevo. *MAXConexiones* es una constante global que indica una capacidad máxima de almacenamiento de 100 conexiones solamente. Dentro del esquema *P* se incluye la restricción correspondiente.

El esquema *P* no se encuentra normalizado porque incluye una declaración y no una firma; es decir, dentro de las declaraciones agrega restricciones que deben ser colocadas en la sección de predicados. Para su normalización, se debe cambiar la secuencia de conexiones (*seq CONEXIÓN)* por su tipo correspondiente y agregar la respectiva restricción en el apartado de predicados. *Seq CONEXION* es realmente una función del tipo $\mathbb{P}(\mathbb{Z} \times CONEXION)$. Entonces, el mismo esquema *P* ahora normalizado y con su nombre cambiado a *Pool* es el siguiente:

$$
\begin{array}{|l}
\hline
\textit{Pool} \\
\hline
conDis : \mathbb{P}(\mathbb{Z} \times CONEXION) \\
\hline
\#conDis \leqslant MAXConexiones \\
conDis \in seq\ CONEXION \\
\hline
\end{array}
$$

El esquema para denotar el estado vacio inicial del agrupamiento antes de cualquier operación es el siguiente:

$$
\begin{array}{|l}
\hline
\textit{InitPool} \\
\hline
\Delta Pool \\
\hline
conDis' = \langle\ \rangle \\
\hline
\end{array}
$$

11.3. Obteniendo conexiones del agrupamiento

Cuando se intenta obtener una conexión del agrupamiento puede haber dos posibles situaciones, cada una se va a modelar de manera separada.

La primera situación es cuando se intenta obtener una conexión del agrupamiento, y en éste hay conexiones disponibles para reutilizar. En este caso, la operación tiene éxito: el elemento más antiguo se entrega como resultado y se elimina del agrupamiento; el estado de la operación se indica también como resultado:

$$
\begin{array}{|l}
\hline
\textit{ObtenerConexionOk} \\
\hline
\Delta Pool \\
resultado! : CONEXION \\
estado! : STATUS \\
\hline
conDis \neq \langle\ \rangle \\
resultado! = head\ conDis \\
conDis' = tail\ conDis \\
estado! = OKConexionObtenida \\
\hline
\end{array}
$$

El segundo caso es cuando se intenta obtener una conexión del pool, pero no hay ninguna disponible para reutilizar en ese momento. En este caso, la operación falla, se indica un estado de error como resultado y no hay cambio en el conjunto de las conexiones que tiene el agrupamiento:

$$
\begin{array}{|l}
\hline
\textit{ObtenerConexionNoOk} \\
\Xi Pool \\
estado! : STATUS \\
\hline
conDis = \langle\, \rangle \\
estado! = ErrorNoHay \\
\hline
\end{array}
$$

11.4. Agregando conexiones al agrupamiento

Al intentar colocar una conexión en el pool, también hay dos posibles escenarios que se modelarán separadamente.

En el primer escenario se intenta colocar una conexión en el agrupamiento, y aún hay espacio disponible para almacenarla; por lo tanto, la conexión se pone al final de la cola actual, y se indica como resultado un estado exitoso en la operación.

$$
\begin{array}{|l}
\hline
\textit{PonerConexionOk} \\
\Delta Pool \\
con? : CONEXION \\
estado! : STATUS \\
\hline
\#conDis < MAXConexiones \\
conDis' = conDis \frown \langle\, con? \,\rangle \\
estado! = OKConexionAgregada \\
\hline
\end{array}
$$

En el segundo escenario también se intenta colocar una conexión en el agrupamiento, pero ya no hay espacio disponible para almacenarla pues ya se alcanzó el número de elementos especificado en *MAXConexiones*. En este caso, no se modifica el estado del agrupamiento, pero se indica un error en el resultado de la operación.

$$
\begin{array}{|l}
\hline
\textit{PonerConexionNoOk} \\
\Xi Pool \\
con? : CONEXION \\
estado! : STATUS \\
\hline
\#conDis = MAXConexiones \\
estado! = ErrorMaxAlcanzado \\
\hline
\end{array}
$$

Resumiendo, las operaciones para obtener y poner conexiones en el agrupamiento se representarían de la siguiente manera:

$$ObtenerConexion \mathrel{\widehat{=}} ObtenerConexionOk \lor ObtenerConexionNoOk$$
$$PonerConexion \mathrel{\widehat{=}} PonerConexionOk \lor PonerConexionNoOk$$

Sin embargo, para propósito de este ejemplo, se van a conservar y considerar los cuatro esquemas individuales de cada escenario mostrados anteriormente.

11.5. Modelando separadamente una bitácora de actividades

Ahora se desea agregar a los cuatro escenarios anteriores, un registro de bitácora para tener una evidencia de cada operación realizada. Se modelará la bitácora aisladamente para después relacionarla con cada uno de los casos utilizando una regla de composición.

Para especificar esta funcionalidad se definen dos tipos: uno para los textos que describen la actividad, la fecha y la hora y el otro para el contenedor mismo de la bitácora (el conjunto completo de entradas):

$$[TEXTO]$$
$$BITACORA == \mathbb{P}\,(TEXTO \times TEXTO)$$

La operación de agregar un registro a la bitácora se puede modelar de la siguiente manera:

AgregarABitacora

$$NombreBitacora? : TEXTO$$
$$Fecha? : TEXTO$$
$$Actividad? : TEXTO$$
$$a! : BITACORA$$
$$obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\rightarrow BITACORA$$

$$\exists_1\, x{:}BITACORA \bullet x = obtenerBitacora(\, NombreBitacora? \,) \land x \neq \varnothing$$
$$a! = obtenerBitacora(\, NombreBitacora? \,) \cup \{ Fecha? \mapsto Actividad? \}$$

Como datos de entrada se requieren: el nombre del contenedor de la bitácora, que puede ser el nombre de un archivo, una base de datos, o una estructura en memoria, un texto que incluya la fecha/hora de registro y una descripción de la actividad realizada. En la especificación se indica: asegurarse de la existencia del contenedor, acceder a él, formar

una nueva entrada con los datos proporcionados y agregarla al conjunto de registros existentes.

11.6. Agregando el registro de bitácora a las operaciones del agrupamiento de conexiones

Para los siguientes sub-apartados, se considerará que cada esquema presentado en los apartados anteriores ha sido capturado en un requerimiento con el mismo nombre y contenido de ese esquema, con la finalidad de introducir reglas de composición cuyos requerimientos involucrados sean fácilmente identificables con los esquemas presentados.

Agregar un registro de bitácora a cada operación del agrupamiento de conexiones es un requerimiento que claramente afecta de igual manera a los cuatro requerimientos del agrupamiento; por eso se dice que es transversal, así que puede especificarse de manera separada como se hizo ya en el apartado 11.5 y posteriormente combinarse con cada uno de los cuatro requerimientos del agrupamiento mediante una regla de composición para cada caso. Para esto, se van a crear cuatro reglas de composición como se muestran en los siguientes apartados.

11.6.1. Registrando en la bitácora la obtención correcta de una conexión

Se agregará el registro de bitácora al requerimiento de *ObtenerConexionOk* mediante la siguiente regla de composición:

```
@COMPOSITIONRULE
@ID “Regla1”;
@RULE
AgregarABitacora ORIG @CC ObtenerConexionOk ORIG
                        @WHERE AFT
                        @CONN AND
                        @CHG  (
                                NombreBitacora?=WRKVAR;
                                Fecha?=WRKVAR;
                                Actividad?=WRKVAR;
                               )
                        @REP  (
                                “Fecha = ‘FechaYHoraActual’";
                                “Actividad = ‘Se ha obtenido
                                exitosamente una conexión del pool’";
                                “NombreBitacora = ‘Bitacora.log’";
                               )
```

Para esta regla se ha utilizado `AFT` para indicar que los elementos de *AgregarBitacora* se deben colocar después de los elementos de *ObtenerConexionOk* en el nuevo requerimiento compuesto. También se ha utilizado el conector lógico `AND` para unirlos a ambos.

Al momento de unir los requerimientos, se deben realizar algunos cambios para que los tipos, las entradas y salidas de ambos sean consistentes. *AgregarABitacora* requiere tres entradas: *NombreBitacora?*, *Fecha?* y *Actividad?*. Sin embargo, al momento de la composición, esos valores deben ser asignados como fijos y no deben representar entradas adicionales sino variables de trabajo internas en el requerimiento compuesto. Para este propósito se utilizan en combinación las secciones `@CHG` y `@REP`: la primera (`@CHG`) especifica que las tres variables dejan de ser de entrada y ahora serán de trabajo. La segunda (`@REP`) les asigna valores fijos a esas mismas variables. Para estas cuatro reglas de composición se van a considerar los mismos valores fijos indicados en la cláusula `@REP` y se trabajará con las versiones originales de los requerimientos (no tendría sentido hacerlo de otra manera, pues en este ejemplo cada requerimiento es atravesado por un solo requerimiento).

11.6.2. Registrando en la bitácora el intento fallido de obtención de una conexión

La segunda regla para agregar un registro de bitácora al intentar obtener una conexión cuando no hay alguna disponible es la siguiente:

```
@COMPOSITIONRULE
@ID “Regla2”;
@RULE
AgregarABitacora ORIG   @CC   ObtenerConexionNoOk ORIG
                        @WHERE BEF
                        @CONN AND
                        @NEWNAME “ObtenerConexionConBitacoraNoOk”
                        @CHG  (
                                Actividad?=WRKVAR;
                                Fecha?=WRKVAR;
                                NombreBitacora?=WRKVAR;
                              )
                        @REP (
                                “Actividad = ‘Error. Se intentó
                                obtener una conexión pero no había
                                ninguna disponible’";
                                “NombreBitacora = ‘Bitacora.log’";
                                “Fecha = ‘FechaYHoraActual’";
                             )
```

Esta segunda regla es similar a la anterior pero tiene dos notables diferencias: en la cláusula `@WHERE` se indica que los elementos de *AgregarBitacora* deben colocarse antes de los elementos de *ObtenerConexionNoOk* en el requerimiento compuesto. Además, se utiliza la cláusula `@NEWNAME` para indicar que el resultado de la composición será un requerimiento con el nombre de *ObtenerConexionConBitacoraNoOk*.

11.6.3. Registrando en la bitácora la colocación exitosa de una conexión

La tercera regla de composición se encarga de agregar una entrada a la bitácora al colocar una conexión exitosamente en el agrupamiento. Esta regla es muy similar a las dos anteriores y se muestra a continuación:

```
@COMPOSITIONRULE
@ID "Regla3";
@RULE
AgregarABitacora ORIG   @CC PonerConexionOk ORIG
                        @WHERE AFT
                        @CONN AND
                        @CHG  (
                                NombreBitacora?=WRKVAR;
                                Actividad?=WRKVAR;
                                Fecha?=WRKVAR;
                              )
                        @REP (
                                "Actividad = 'Se ha agregado
                                exitosamente una conexión al pool.'";
                                "NombreBitacora = 'Bitacora.log'";
                                "Fecha = 'FechaYHoraActual'";
                             )
```

En esta tercera regla no se especifica un nuevo nombre para el requerimiento compuesto, por lo que éste conservará el nombre de *PonerConexionOk*. También se indica que los elementos de *AgregarBitacora* se colocan después de los de *PonerConexionOk*.

11.6.4. Registrando en la bitácora el intento fallido de colocación de una conexión

La cuarta y última regla de composición del ejemplo es la encargada de agregar una entrada a la bitácora cuando se intenta colocar una conexión en el pool, pero éste ha alcanzado ya su máxima capacidad y no tiene espacio adicional. Esta regla tiene la misma estructura que sus predecesoras y se muestra a continuación:

```
@COMPOSITIONRULE
@ID "Regla4";
@RULE
AgregarABitacora ORIG @CC PonerConexionNoOk ORIG
                        @WHERE BEF
                        @CONN AND
                        @NEWNAME "PonerConexionConBitacoraNoOk"
                        @CHG  (
                                NombreBitacora?=WRKVAR;
                                Actividad?=WRKVAR;
                                Fecha?=WRKVAR;
                              )
                        @REP (
                                "Actividad = 'ERROR.Se ha intentado
                                colocar una conexión en el pool, pero
                                ya está lleno.'";
                                "Fecha = 'FechaYHoraActual'";
                                "NombreBitacora = 'Bitacora.log'";
                             )
```

11.7. Ejemplo de ejecución del algoritmo de composición

Una vez que se tiene la especificación completa de los requerimientos, intereses y reglas, ya es posible generar el modelo completo mediante la ejecución del algoritmo de composición de la Figura 8.4.

Con el objetivo de mostrar un ejemplo de la ejecución del algoritmo, se tomará la especificación realizada para el ejemplo del presente capítulo y se mostrará detalladamente el proceso de construcción de la primera parte del modelo final, la cual es el resultado de tomar la primera regla y actuar en consecuencia para construir el respectivo requerimiento compuesto.

El algoritmo comienza con un modelo completo vacío, y continúa leyendo la primera regla de composición. Lee el primer y segundo requerimiento involucrados en dicha regla, los cuales son *AgregarABitacora*, y *ObtenerConexionOk* en su versión original (en este caso, es la única opción posible, pues aún no hay ninguna versión compuesta disponible).

Posteriormente, se construye un nuevo requerimiento vacío. En la regla de este ejemplo, no se ha especificado ningún nombre para el nuevo requerimiento, por lo que el algoritmo le asigna el identificador de *ObtenerConexionOk*, que es el mismo nombre del requerimiento que fue atravesado.

En este último requerimiento recién creado, se insertan los elementos de ambos requerimientos en las secciones de declaraciones y condiciones de acuerdo a los valores asociados a las cláusulas `@WHERE`, `@CONN` y `@COND`. La cláusula `@WHERE` indica que *AgregarABitacora* debe afectar a *ObtenerConexionOk* después (`AFT`) del contenido original de éste último. Mediante `@CONN` se sabe que la relación establecida es del tipo

conjunción (AND). Para éste ejemplo en particular no hay valor para la cláusula @COND, por lo que no hay un predicado adicional que se deba agregar durante la composición.

En este punto, el requerimiento compuesto toma la siguiente forma:

ObtenerConexionOk

$\Delta Pool$
$resultado! : CONEXION$
$estado! : STATUS$
$NombreBitacora? : TEXTO$
$Fecha? : TEXTO$
$Actividad? : TEXTO$
$a! : BITACORA$
$obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\rightarrow BITACORA$

$($
$conDis \neq \langle\, \rangle$
$resultado! = head\ conDis$
$conDis' = tail\ conDis$
$estado! = OKConexionObtenida$
$)$
$\wedge($
$\exists_1\, x{:}BITACORA \bullet x = obtenerBitacora(\, NombreBitacora?\,) \wedge x \neq \varnothing$
$a! = obtenerBitacora(\, NombreBitacora?\,) \cup \{Fecha? \mapsto Actividad?\, \})$

En esta regla de composición, la ausencia de la cláusula @ADD indica que no hay nuevas declaraciones para agregar, por lo que no se hace ningún cambio en este sentido.

El siguiente paso es tomar la cláusula @CHG y realizar el cambio de naturaleza de las variables indicadas. Para este ejemplo, la regla indica que se deben realizar tres cambios: Las variables *NombreBitacora?*, *Fecha?* y *Actividad?* pasan a ser todas variables de trabajo. En este punto, el requerimiento compuesto sufre las siguientes modificaciones:

*ObtenerConexionOk*

$\Delta Pool$
$resultado!\ : CONEXION$
$estado! : STATUS$
$NombreBitacora : TEXTO$
$Fecha : TEXTO$
$Actividad : TEXTO$
$a! : BITACORA$
$obtenerBitacora: TEXTO \rightarrowtail\!\!\!\rightarrow BITACORA$

(
$conDis \neq \langle\,\rangle$
$resultado! = head\ conDis$
$conDis' = tail\ conDis$
$estado! = OKConexionObtenida$
)
^(
$\exists_1\ x{:}BITACORA \bullet x = obtenerBitacora(\ NombreBitacora\ )\ \wedge\ x \neq \emptyset$
$a! = obtenerBitacora(\ NombreBitacora\ ) \cup \{Fecha \mapsto Actividad\ \})$

Ahora, se toman los datos incluidos en la cláusula `@REP` para realizar la sustitución de los valores de entrada correspondientes. La regla indica tres cambios: La variable *Fecha* toma el valor de "*FechaYHoraActual*", a la variable *Actividad* se le asigna el valor de "*Se ha obtenido exitosamente una conexión del pool*" y la variable *NombreBitacora* adquiere el valor de "*Bitacora.log*". La representación del requerimiento compuesto con estos cambios se muestra a continuación:

ObtenerConexionOk

$\Delta Pool$
$resultado! : CONEXION$
$estado! : STATUS$
$NombreBitacora : TEXTO$
$Fecha : TEXTO$
$Actividad : TEXTO$
$a!: BITACORA$
$obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\!\rightarrow BITACORA$

(
$conDis \neq \langle \rangle$
$resultado! = head \; conDis$
$conDis' = tail \; conDis$
$estado! = OKConexionObtenida$
)
^(
$\exists_1 \, x{:}BITACORA \bullet x = obtenerBitacora(NombreBitacora) \wedge x \neq \emptyset$
$a! = obtenerBitacora(NombreBitacora) \cup \{Fecha \mapsto Actividad \}$
)
Fecha = "FechaYHoraActual"
Actividad = "Se ha obtenido exitosamente una conexión del pool"
NombreBitacora = "Bitacora.log"

De esta manera el requerimiento queda almacenado en el modelo compuesto, y como la regla de este ejemplo no incluye el operador asterisco (*), el algoritmo continúa examinando las siguientes reglas de composición, repitiendo el mismo proceso aquí desglosado para cada una de ellas.

El modelo compuesto completo que resulta al finalizar la ejecución del algoritmo se muestra y se analiza en el apartado 11.8.

11.8. El modelo compuesto completo

Una vez que las reglas han sido especificadas, y que se ha generado el modelo completo, se obtienen los siguientes cuatro esquemas:

ObtenerConexionOk

$\Delta Pool$

$resultado! : CONEXION$

$estado! : STATUS$

$NombreBitacora : TEXTO$

$Fecha : TEXTO$

$Actividad : TEXTO$

$a! : BITACORA$

$obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\!\!+ BITACORA$

$($

$conDis \neq \langle\, \rangle$

$resultado! = head\ \ conDis$

$conDis' = tail\ \ conDis$

$estado! = OKConexionObtenida$

$)$

$\wedge($

$\exists_1\, x{:}BITACORA \bullet x = obtenerBitacora(\, NombreBitacora\,) \ \wedge\ x \neq \varnothing$

$a! \ = \ obtenerBitacora(\, NombreBitacora\,) \ \cup \{Fecha \mapsto Actividad\,\}$

$)$

$Fecha = "FechaYHoraActual"$

$Actividad = "Se\ ha\ obtenido\ exitosamente\ una\ conexión\ del\ pool"$

$NombreBitacora = "Bitacora.log"$

ObtenerConexionConBitacoraNoOk

$\Xi Pool$

$NombreBitacora : TEXTO$

$Fecha : TEXTO$

$Actividad : TEXTO$

$a! : BITACORA$

$obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\rightarrow BITACORA$

$estado! : STATUS$

$($

$\exists_1\, x{:}BITACORA \bullet x = obtenerBitacora(\, NombreBitacora\,) \;\wedge\; x \neq \varnothing$

$a! \;=\; obtenerBitacora(\, NombreBitacora\,) \;\cup\; \{Fecha \mapsto Actividad\,\}$

$)$

$\wedge($

$conDis = \langle\, \rangle$

$estado! = ErrorNoHay$

$)$

$Actividad = \text{"Error. Se intentó obtener una conexión pero no había ninguna disponible"}$

$NombreBitacora = \text{"Bitacora.log"}$

$Fecha = \text{"FechaYHoraActual"}$

PonerConexionOk

$\Delta Pool$

$con? : CONEXION$

$estado! : STATUS$

$NombreBitacora : TEXTO$

$Fecha : TEXTO$

$Actividad : TEXTO$

$a! : BITACORA$

$obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\rightarrow BITACORA$

$($

$\#conDis < MAXConexiones$

$conDis' = conDis \frown \langle con? \rangle$

$estado! = OKConexionAgregada$

$)$

$\wedge($

$\exists_1\, x{:}BITACORA \bullet x = obtenerBitacora(\, NombreBitacora\,) \wedge x \neq \varnothing$

$a! = obtenerBitacora(\, NombreBitacora\,) \cup \{Fecha \mapsto Actividad\}$

$)$

$Actividad = \text{"Se ha agregado exitosamente una conexión al pool."}$

$NombreBitacora = \text{"Bitacora.log"}$

$Fecha = \text{"FechaYHoraActual"}$

$$
\begin{array}{l}
\underline{\textit{PonerConexionConBitacoraNoOk}} \\
\Xi Pool \\
NombreBitacora : TEXTO \\
Fecha : TEXTO \\
Actividad : TEXTO \\
a! : BITACORA \\
obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\rightarrow BITACORA \\
con? : CONEXION \\
estado! : STATUS \\
\hline
(\\
\exists_1\, x{:}BITACORA \bullet x = obtenerBitacora(\, NombreBitacora\,) \ \wedge\ x \neq \varnothing \\
a! \ = \ obtenerBitacora(\, NombreBitacora\,) \ \cup \{Fecha \mapsto Actividad\,\} \\
) \\
\wedge(\\
\#conDis = MAXConexiones \\
\ estado! = ErrorMaxAlcanzado \\
) \\
Actividad = \text{"ERROR.Se ha intentado colocar una conexión en el pool, pero ya está lleno."} \\
Fecha = \text{"FechaYHoraActual"} \\
NombreBitacora = \text{"Bitacora.log"}
\end{array}
$$

En estos cuatro esquemas compuestos es importante notar la correspondencia de sus características individuales con los parámetros especificados en la regla de composición de la cual surgieron.

Por citar algunos puntos sobresalientes, obsérvese como el primer y tercer requerimiento compuesto mantienen el nombre del segundo operador de su regla asociada (*ObtenerConexionOk* y *PonerConexionOk* respectivamente) pues no se especificó otro nombre usando la cláusula `@NEWNAME`. Por el contrario, el segundo y cuarto requerimiento compuesto tienen los nombres que se indicaron en sus respectivas reglas de origen (*ObtenerConexionConBitacoraNoOk* y *PonerConexionConBitacoraNoOk*).

Por otra parte, en el primer y tercer requerimiento compuesto, los elementos de *AgregarBitacora* se colocaron después de los elementos de los requerimientos atravesados como efecto de utilizar la constante `AFT` en la cláusula `@WHERE`. En el segundo y cuarto requerimiento compuesto, los elementos de *AgregarBitacora* fueron colocados antes de los elementos de los requerimientos atravesados como resultado de utilizar el valor `BEF` en la cláusula `@WHERE`.

También es importante resaltar que en los requerimientos compuestos, como resultado de utilizar la cláusula `@CHG` en las reglas, las variables *Archivo*, *Fecha* y *Actividad* aparecen como variables de trabajo y no como entradas esperadas. Así mismo, el uso de la cláusula `@REP` provocó que los valores fijos para estas mismas variables fueran asignados en la sección de predicados de cada requerimiento compuesto.

11.9. Conclusiones

En situaciones donde existen requerimientos transversales, el uso de requerimientos aislados y reglas de composición ayuda a promover una buena separación de intereses, la cual a su vez permite mejorar el mantenimiento global de la especificación. Aún en aislamiento, el contenido de los requerimientos y las reglas determinan el modelo compuesto completo de un proyecto. Como el proceso de composición se encuentra guiado por las instrucciones que contienen las reglas, para alterar la estructura y el contenido de los requerimientos compuestos se debe siempre revisar y modificar la especificación de las reglas del proyecto.

Es importante mencionar que los esquemas compuestos incluidos en este trabajo se apegan a la representación de los requerimientos compuestos que SPZS proporciona actualmente; los esquemas mostrados aquí son versiones tipográficamente estilizadas de la representación en texto plano que SPZS hace de ellos.

Para la comprobación de estos esquemas, se utilizó JAZA -Just Another Z Animator- (UTTING, Mark, Julio, 2000), un software gratuito y de libre distribución que permitió animar (ejecutar) las especificaciones para analizar su comportamiento. Durante este proceso, se hicieron algunas consideraciones principalmente debidas a la adecuación al uso y a las capacidades de JAZA y al modo de trabajo de SPZS y Z-SIIMD. Por ejemplo, JAZA utiliza una sintaxis predeterminada basada en marcas para representar las especificaciones, la cual es distinta a la notación libre utilizada en estos ejemplos; JAZA tampoco soporta las definiciones axiomáticas y en este ejemplo se incluía una. Por otra parte, SPZS en su versión actual, solamente expresa al final del requerimiento compuesto los cambios de valores que se hacen con la cláusula `@REP` en las reglas, y no realiza la sustitución completa correspondiente en el cuerpo del requerimiento completo, como JAZA lo esperaría.

Como resultado final, el proceso de animación y análisis de las especificaciones del ejemplo de este capítulo con JAZA demostró que el proceso de composición se encuentra bien encaminado, pues produjo requerimientos compuestos correctos y coherentes.

Capítulo 12.
Aplicación de los procesos de clasificación e instanciación.
Caso de uso: Retiro bancario

12.1. Introducción

En este capítulo se enfatizará la aplicación de los algoritmos de clasificación e instanciación haciendo referencia al prototipo de software SPZS descrito en el Capítulo 10. Para abordar la aplicación del algoritmo de clasificación, se asumirá que la especificación completa del caso de uso del Capítulo 11 fue capturada utilizando SPZS y que se desea reutilizar el interés relacionado con la inclusión de un registro de bitácora. Una vez almacenado este interés en el repositorio de elementos reutilizables, se creará un nuevo proyecto en blanco para realizar la especificación de dos escenarios en un retiro bancario. Desde ahí se abordará el proceso de instanciación; primero se va a realizar la búsqueda de los elementos relacionados a una categoría específica, luego se seleccionarán los indicados y finalmente se incorporarán al nuevo proyecto.

12.2. Fase de clasificación

Suponer que la especificación del ejemplo tratado en el Capítulo 11 se organiza en forma de intereses tal como se muestra en la Figura 12.1. En este apartado se enfocará la atención en el interés llamado "*Bitacora*", ya que es el que se piensa enviar al repositorio para posteriormente incorporarlo en otro proyecto. El interés denominado "*Bitacora*" ha sido capturado con los detalles mostrados en la Figura 12.2, éstos incluyen las categorías de "*almacenamiento*" y "*no funcional*", así como las palabras clave "*registro*", "*bitácora*" y "*bitacora*". Dentro de este interés se encuentran dos requerimientos: uno que describe los tipos necesarios para la especificación y otro que contiene la especificación para agregar un nuevo registro a la bitácora. Ambos requerimientos se explicaron previamente en el apartado 11.5.

Si el desarrollador desea enviar el interés "*Bitacora*" al repositorio de elementos, debe seleccionarlo como se muestra en la Figura 12.1 y oprimir el botón marcado con la letra "R". En ese momento, SPZS toma el interés en cuestión, valida que tenga datos de clasificación asociados, muestra los datos actuales en pantalla para que sean revisados y solicita la confirmación del desarrollador, tal como se observa en la Figura 12.3; todo esto sucede antes de enviarlo al repositorio de intereses.

Si el desarrollador, al observar los datos decide hacer algún cambio de último momento, puede cancelar el proceso en ese instante desde esa misma ventana de revisión, contestando negativamente a la pregunta de confirmación que el software le hace. Por el contrario, si se

contesta afirmativamente, el algoritmo determina el repositorio en el cual se va a guardar el elemento (en este caso, es el de los intereses) y lo almacena ahí.

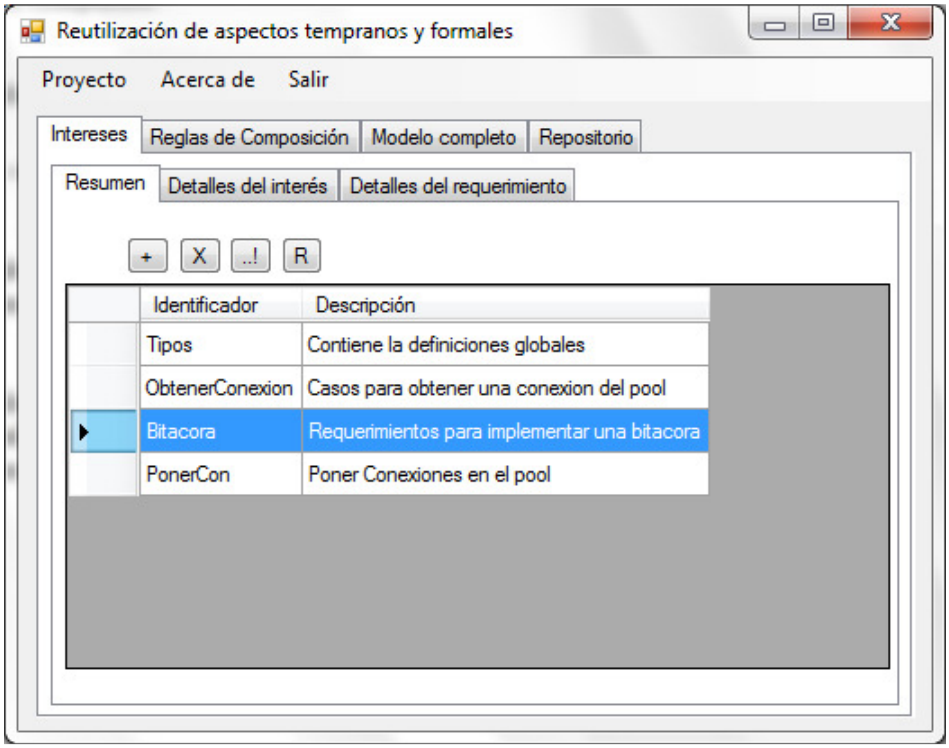


Figura 12.1 Vista general de los intereses del ejemplo.

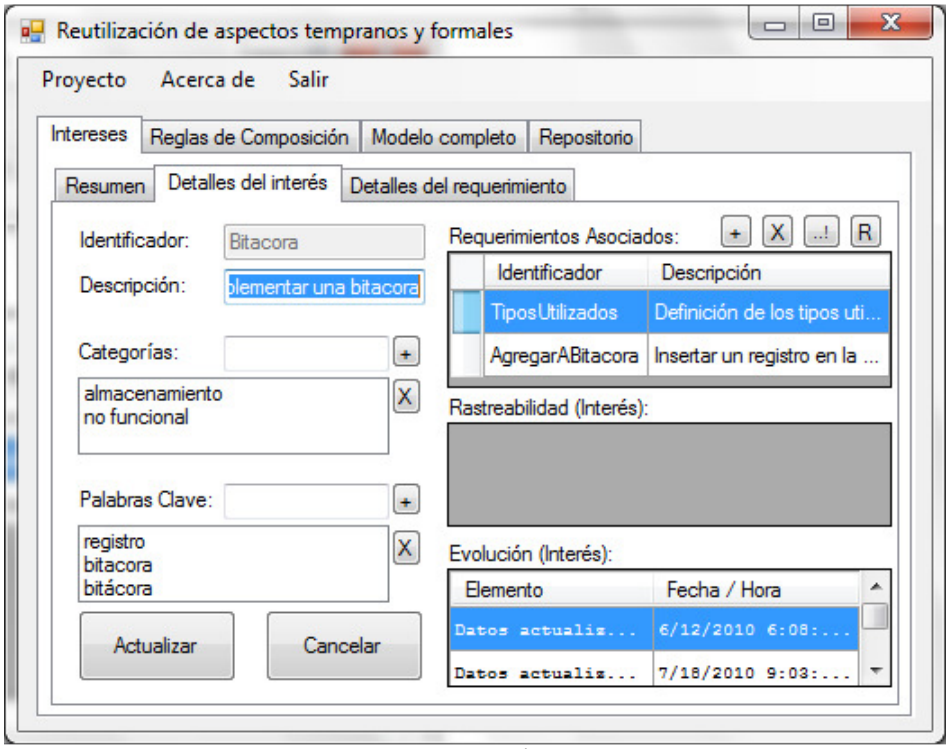


Figura 12.2 Detalles del interés llamado Bitacora.

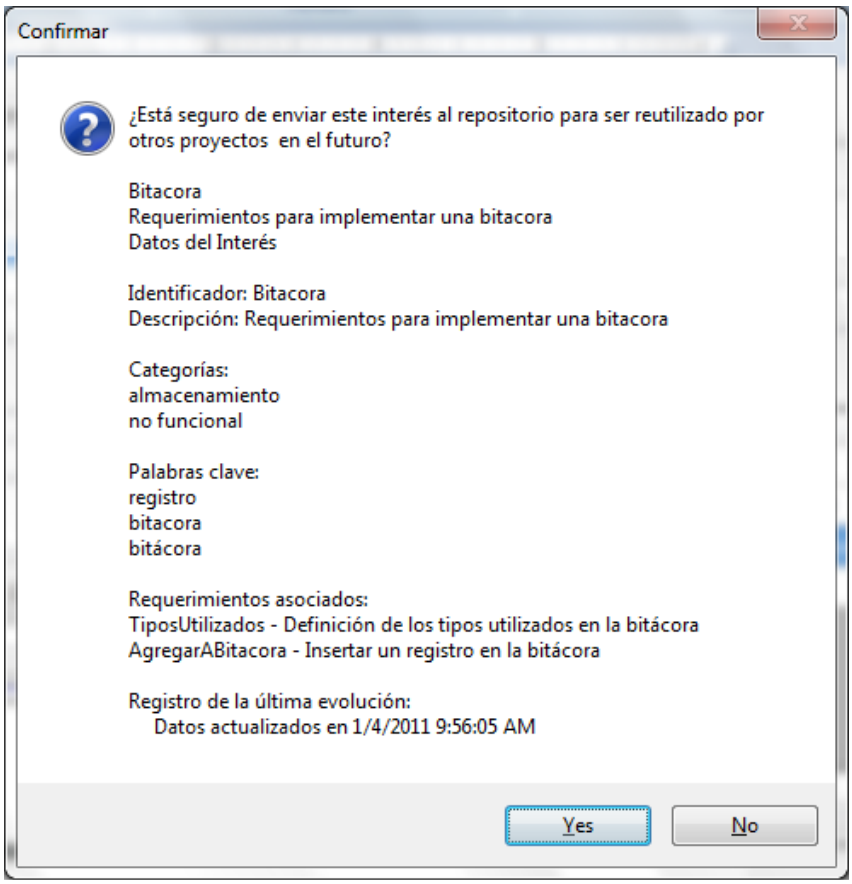


Figura 12.3 Confirmación para almacenar el interés en el repositorio.

12.3. Fase de instanciación (en un proyecto diferente)

Suponer ahora que se iniciará un nuevo proyecto en blanco en el cual se modelarán dos posibles casos de retiro de dinero de una cuenta bancaria. En el primer caso, el retiro de dinero se realiza correctamente pues la cantidad solicitada es menor o igual al saldo actual de la cuenta. En el segundo escenario, el retiro no se puede llevar a cabo debido a que la cantidad solicitada supera los fondos disponibles de la cuenta en ese momento. El resultado de cada operación se debe informar siempre al término de las transacciones. Este caso de uso se retoma en parte, del ejemplo mostrado en el apartado 14.17.1.

Se comienza definiendo un tipo para los posibles resultados de las operaciones:

RESULTADO :: = RetiroExitoso | *SaldoInsuficiente*

Para definir el estado del sistema se realiza un esquema llamado *CuentaBancaria* que contiene la definición y las restricciones para el saldo, el cual debe ser siempre un número entero positivo o cero.

CuentaBancaria

$saldo : \mathbb{Z}$

$saldo \geqslant 0$

Luego el estado inicial coloca el saldo de la cuenta a cero:

InitCuentaBancaria

$\Delta CuentaBancaria$

$saldo' = 0$

Para especificar el primer escenario que se desea modelar, donde el retiro se realiza con éxito, se utiliza un esquema llamado *RetiroCorrecto*, el cual recibe como entrada el valor de *cantidadARetirar?* que representa la cantidad solicitada para el retiro, y entrega como salida *res!*, que representa el resultado de la operación, exitosa en este caso.

RetiroCorrecto

$\Delta CuentaBancaria$

$cantidadARetirar? : \mathbb{Z}$

$res! : RESULTADO$

$cantidadARetirar? > 0$

$cantidadARetirar? \leqslant saldo$

$saldo' = saldo - cantidadARetirar?$

$res! = RetiroExitoso$

Ahora, se especifica el segundo caso, donde el saldo en ese momento no es suficiente para cubrir la cantidad que se desea retirar; de esta manera, el saldo permanece intacto y se indica como resultado un estado de saldo insuficiente. Todo esto se incluye en el esquema *RetiroIncorrecto*.

RetiroIncorrecto

$\Xi CuentaBancaria$

$cantidadARetirar? : \mathbb{Z}$

$res! : RESULTADO$

$cantidadARetirar? > 0$

$cantidadARetirar? > saldo$

$res! = SaldoInsuficiente$

En este punto del proceso se utiliza el software SPZS para introducir estas especificaciones en un proyecto vacío (Figura 12.4) los cuales se capturan organizadas en dos intereses (Figura 12.5): uno llamado *Retiro*, agrupando dos requerimientos (*RetiroCorrecto* y *RetiroIncorrecto*), y otro llamado *Datos* con la especificación de los tipos, la cuenta bancaria y su estado inicial.

Ahora, suponer que se desea agregar una funcionalidad de registro de bitácora a ambos escenarios de retiro. En este momento, se tienen dos opciones: se puede comenzar a trabajar en esa especificación partiendo desde cero, o bien, se puede buscar y reutilizar una que ya exista en el repositorio. Como esa especificación ya se capturó en el proyecto anterior (agrupamiento de conexiones) y fue almacenada en el repositorio dentro de un interés, se va a realizar una búsqueda en el repositorio, y se va a incorporar al proyecto actual.

Primero es necesario recurrir al repositorio para buscar los elementos disponibles ya sea por identificador, descripción, categoría o palabra clave. También se puede obtener una lista que contenga todos los elementos existentes en ese momento en cualquiera de los tres repositorios: intereses, requerimientos o reglas. Para el caso del presente ejemplo, se realiza una búsqueda en la categoría de "almacenamiento" tal como se observa en la Figura 12.6, obteniendo todos los elementos relacionados con esa categoría.

En el listado que se obtiene, aparece el interés deseado: se trata del interés llamado "*Bitacora*". Para incorporarlo al proyecto actual, el desarrollador debe seleccionarlo de la lista que se obtiene como resultado y oprimir el botón etiquetado con la palabra "Instanciar". Cuando esto sucede, el desarrollador debe adaptar el nombre del interés para que se adecúe a la nomenclatura utilizada, al mismo tiempo que se asegura que no hay una duplicidad de identificadores en los elementos del actual proyecto. Para este caso particular, el identificador del interés se mantiene sin cambio, tal como se observa en la Figura 12.7.

Una vez terminado este proceso, el interés ya forma parte del proyecto actual, tal como se muestra en la Figura 12.8 y puede ser manipulado de la misma manera como si hubiera sido introducido desde cero por el desarrollador.

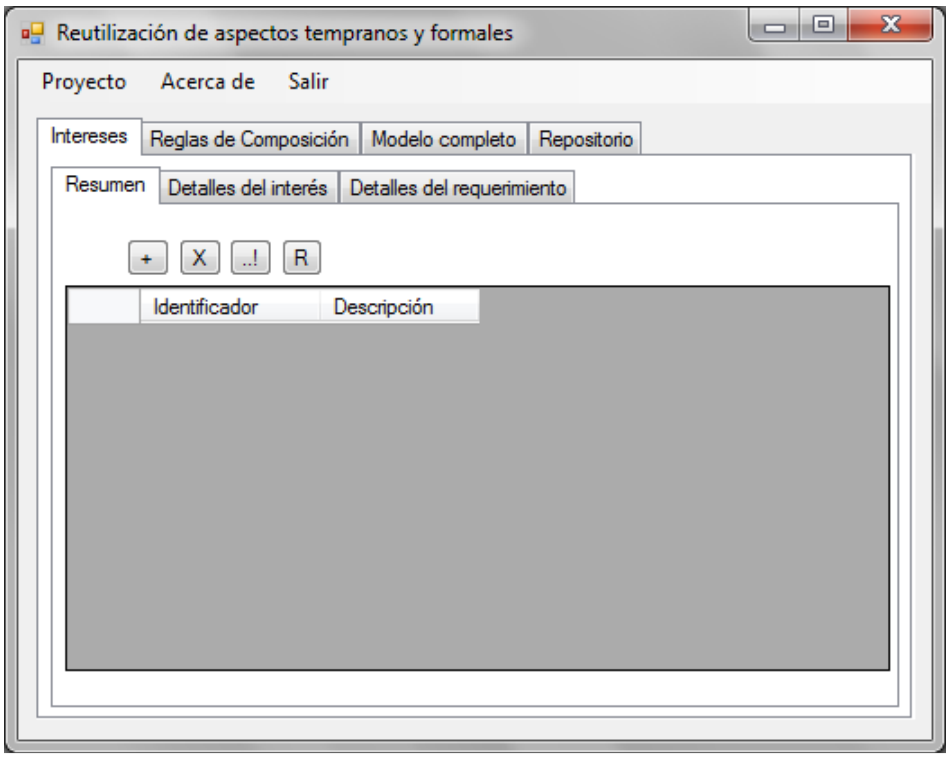


Figura 12.4 Un proyecto de especificación nuevo.

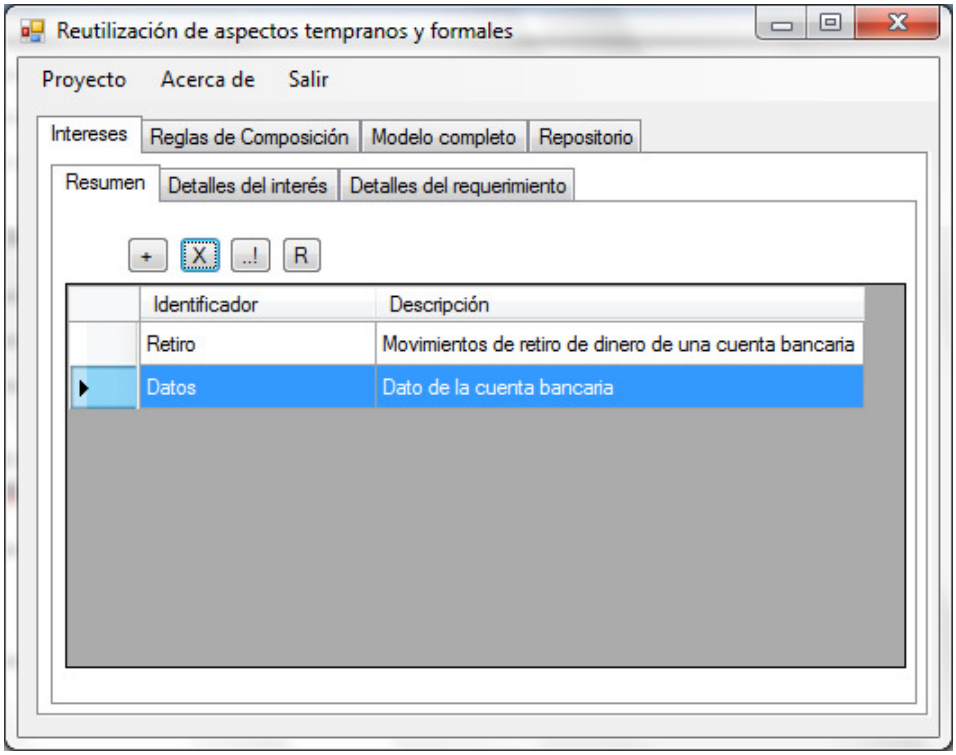


Figura 12.5 Intereses del proyecto.

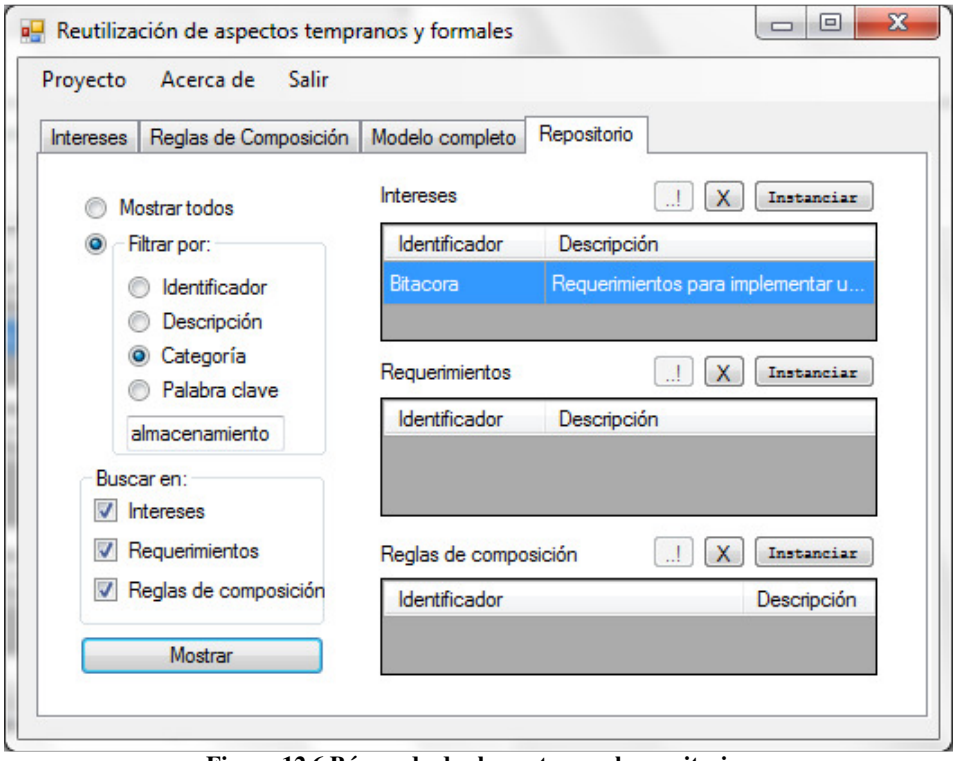


Figura 12.6 Búsqueda de elementos en el repositorio.

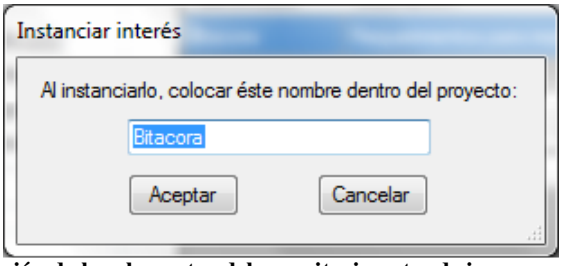


Figura 12.7 Adecuación de los elementos del repositorio antes de incorporarlos a un proyecto.

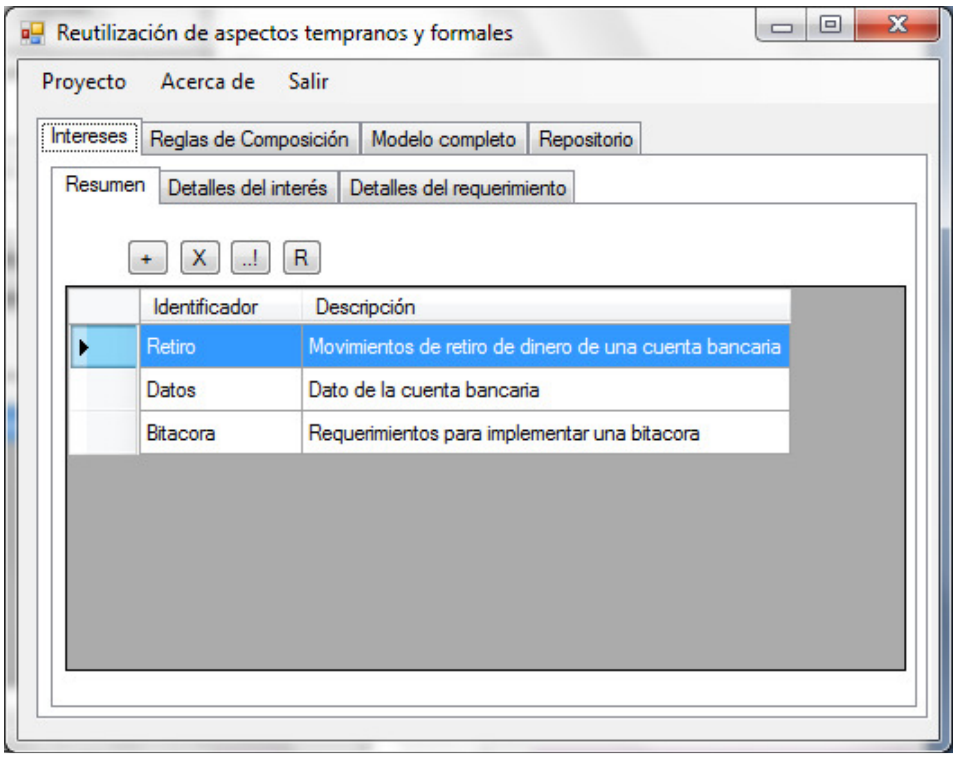


Figura 12.8 Un interés del repositorio incorporado en el nuevo proyecto.

12.4. Agregando registro de bitácora a las operaciones de retiro

Para agregar la especificación de la bitácora a los dos casos de retiro de la cuenta bancaria, se va a escribir una sola regla de composición que afecte a ambos escenarios. En la regla se aprovechará la organización de los dos requerimientos dentro de un interés para utilizar el operador asterisco (*), el cual, en este caso, hace posible que el requerimiento nombrado *AgregarABitacora* atraviese a todos los requerimientos que se encuentran dentro del interés llamado *Retiro*.

En la regla también se incorporará una nueva declaración con la cláusula `@ADDVAR` para introducir *descripcion*, una función que genera un texto en base al resultado de la operación del retiro. Por su parte, la cláusula `@COND` agrega una asignación relacionada con *descripcion* que contiene los posibles resultados de la operación de retiro con sus respectivos textos para incluir en la bitácora.

La especificación completa de la regla de composición se muestra a continuación. La pantalla de captura donde se introducen los datos de la regla se encuentra en la Figura 12.9 y el resumen de la regla dentro del proyecto se presenta en la Figura 12.10.

```
@COMPOSITIONRULE
@ID "BitacoraAlRetiro";
@RULE
Bitacora.AgregarABitacora ORIG
                        @CC   Retiro.* ORIG
                        @WHERE AFT
                        @CONN AND
                        @ADDVAR(
                                "descripcion : RESULTADO ⟶ TEXTO";
                               )
                        @COND  (
                                "descripcion = { RetiroExitoso ↦  'Se
                                ha realizado un retiro exitosamente',
                                SaldoInsuficiente ↦  'Se ha intentado
                                retirar pero no hay saldo suficiente'
                                }";
                               )
                        @CHG   (
                                Fecha?=WRKVAR;
                                Actividad?=WRKVAR;
                                NombreBitacora?=WRKVAR;
                               )
                        @REP   (
                                "Fecha = 'FechaYHoraActual'";
                                "Actividad = descripcion(res!)";
                                "NombreBitacora = 'Bitacora.txt'";
                               )
```

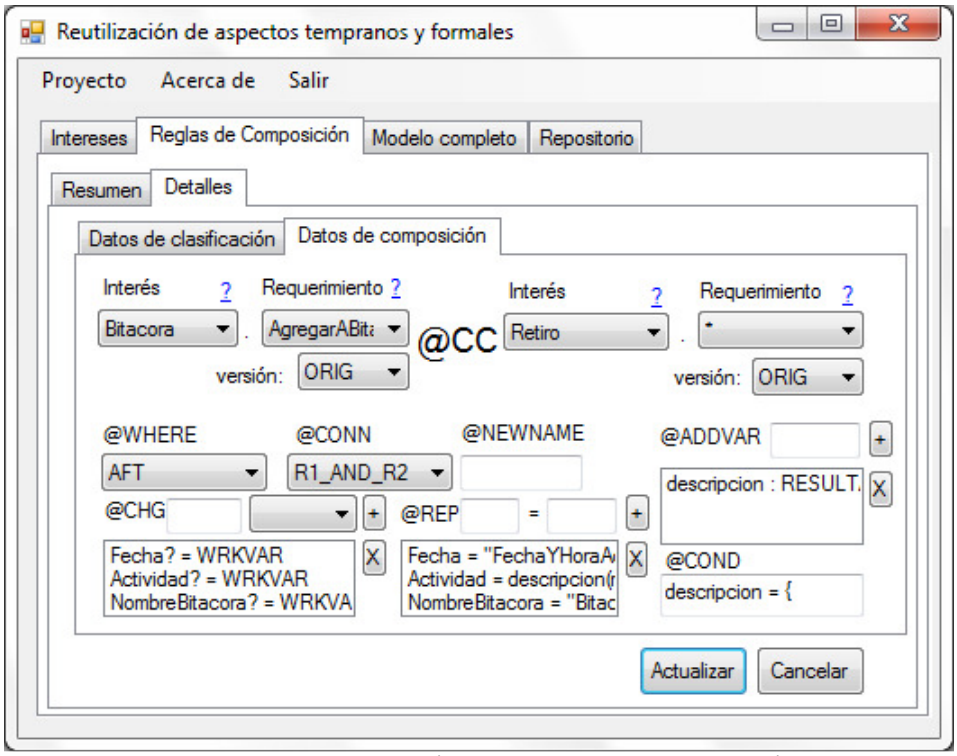


Figura 12.9 Captura de la regla de composición para agregar el registro de bitácora al retiro bancario.

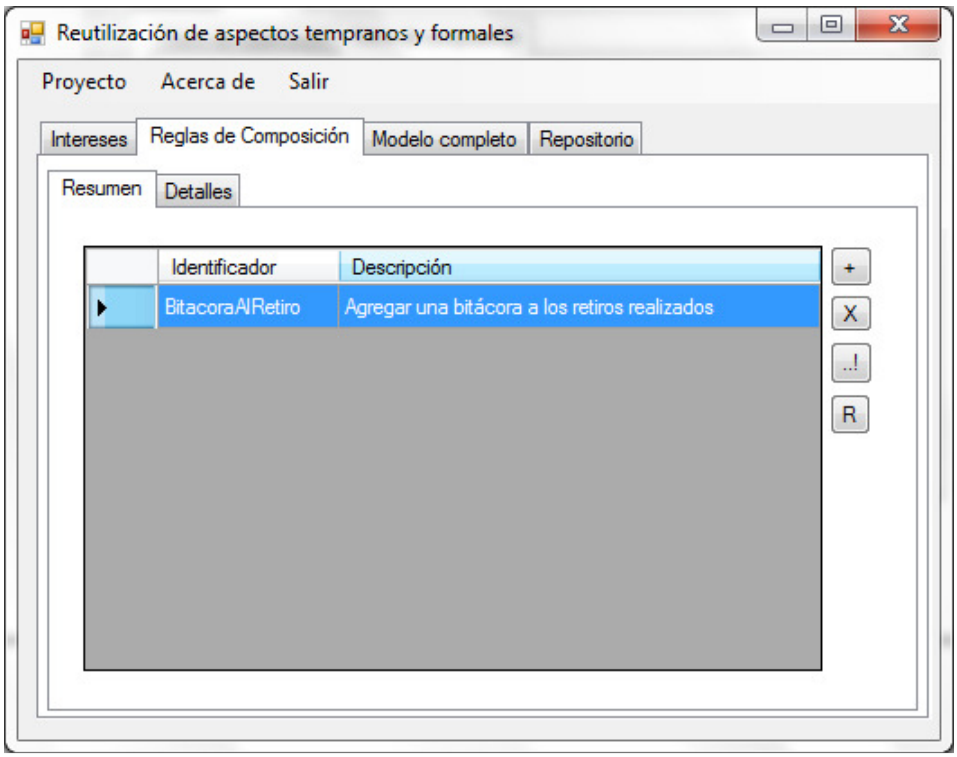


Figura 12.10 Resumen de la regla de composición capturada.

12.5. El modelo compuesto

Después del proceso de composición al que se accede desde el apartado "Modelo completo" de SPZ, los dos requerimientos afectados por el registro de bitácora se presentan en pantalla tal como se muestra en la Figura 12.11. El contenido del requerimiento *RetiroCorrecto* después de ser atravesado por el requerimiento *AgregarABitacora* queda de la siguiente manera:

RetiroCorrecto

$\Delta CuentaBancaria$
$cantidadARetirar? : \mathbb{Z}$
$res! : RESULTADO$
$NombreBitacora : TEXTO$
$Fecha : TEXTO$
$Actividad : TEXTO$
$a! : BITACORA$
$obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\rightarrow BITACORA$
$descripcion : RESULTADO \rightarrowtail TEXTO$

$($
$cantidadARetirar? > 0$
$cantidadARetirar? \leqslant saldo$
$saldo' = saldo - cantidadARetirar?$
$res! = RetiroExitoso$
$)$
$\wedge($
$\exists_1\, x{:}BITACORA \bullet x = obtenerBitacora(\, NombreBitacora\,) \wedge x \neq \varnothing$
$a! = obtenerBitacora(\, NombreBitacora\,) \cup \{Fecha \mapsto Actividad\,\}$
$)$
$Fecha = "FechaYHoraActual"$
$Actividad = descripcion(res!)$
$NombreBitacora = "Bitacora.txt"$
$descripcion = \{RetiroExitoso \mapsto$ *"Se ha realizado un retiro exitosamente"*,
$SaldoInsuficiente \mapsto$ *"Se ha intentado retirar pero no hay saldo suficiente"*$\}$

Después de que el requerimiento *AgregarABitacora* atraviesa al requerimiento *RetiroIncorrecto*, el contenido de éste último queda de la siguiente manera:

$$
\begin{array}{l}
\underline{\textit{RetiroIncorrecto}} \\
\Xi CuentaBancaria \\
cantidadARetirar? : \mathbb{Z} \\
res! : RESULTADO \\
NombreBitacora : TEXTO \\
Fecha : TEXTO \\
Actividad : TEXTO \\
a! : BITACORA \\
obtenerBitacora : TEXTO \rightarrowtail\!\!\!\!\rightarrow BITACORA \\
descripcion : RESULTADO \rightarrowtail TEXTO \\
\hline
(\\
cantidadARetirar? > 0 \\
cantidadARetirar? > saldo \\
res! = SaldoInsuficiente \\
) \\
\wedge (\\
\exists_1\, x{:}BITACORA \bullet x = obtenerBitacora(\, NombreBitacora\,) \wedge x \neq \varnothing \\
a! = obtenerBitacora(\, NombreBitacora\,) \cup \{Fecha \mapsto Actividad\,\} \\
) \\
Fecha = "FechaYHoraActual" \\
Actividad = descripcion(res!) \\
NombreBitacora = "Bitacora.txt" \\
descripcion = \{RetiroExitoso \mapsto \text{“Se ha realizado un retiro exitosamente”}, \\
SaldoInsuficiente \mapsto \text{“Se ha intentado retirar pero no hay saldo suficiente”}\}
\end{array}
$$

Obsérvese como ambos requerimientos compuestos tienen una similitud estructural pues su construcción fue dirigida por la misma regla. También es importante resaltar que una misma regla pudo especificar la influencia de un requerimiento sobre otros dos ya que éstos se encontraban organizados dentro de un mismo interés. En esta ocasión la regla de composición también incluyó las cláusulas `@ADDVAR` y `@COND` para insertar declaraciones e invariantes en los nuevos requerimientos.

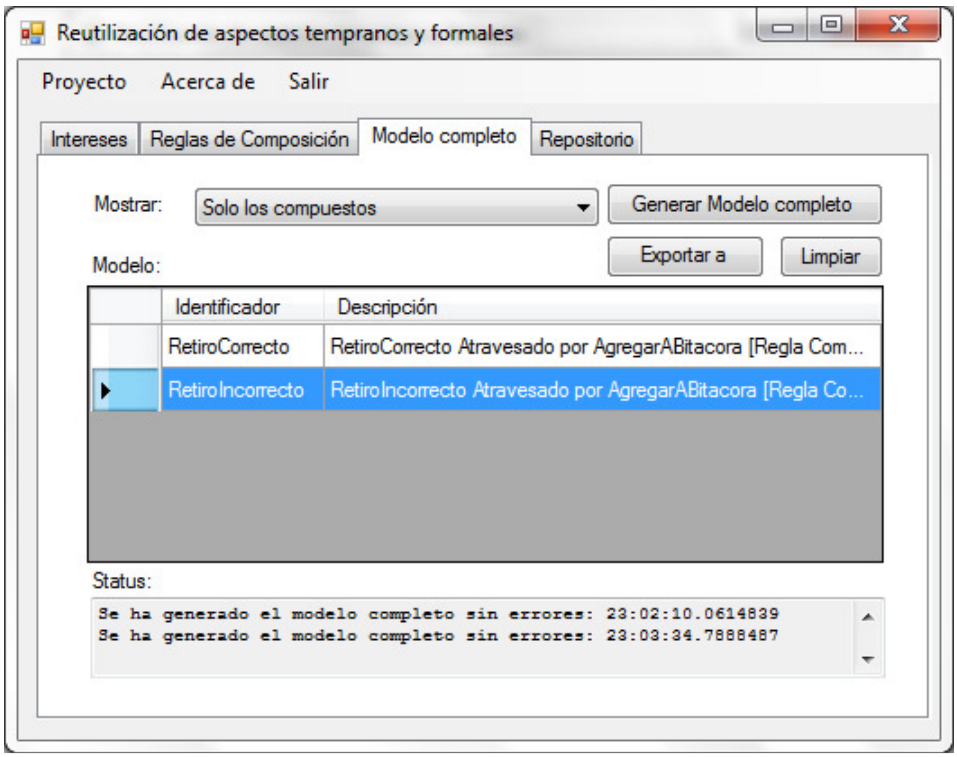


Figura 12.11 Generación del modelo completo en el proyecto de retiro de la cuenta bancaria.

12.6. Conclusiones

En este capítulo se presentó un escenario que muestra el proceso de aplicación de los algoritmos de reutilización e instanciación utilizando SPZ. Con los ejemplos se ha mostrado que un interés incluido en una especificación de un proyecto puede ser reutilizado en otro proyecto en un dominio diferente. También se ha resaltado la ventaja de la organización de los requerimientos dentro de intereses al especificar reglas de composición que deban afectar a más de un requerimiento. Por otra parte, también se puso en relieve la importancia de los repositorios y de la interfaz del software que proporciona acceso a ellos mediante los algoritmos de clasificación, búsqueda e instanciación.

En el modelo actual, las nuevas reglas de composición se deben especificar a partir de cero; sería oportuno poder contar con un algoritmo que permitiera una construcción inicial de ellas de manera semi-automatizada y guiada, tomando como referencia los elementos que están disponibles en la especificación en ese momento y que son candidatos para formar una regla coherente. El algoritmo podría basarse en los resultados de un análisis de compatibilidad entre los requerimientos para ofrecer sugerencias y configuraciones durante el proceso de creación de reglas.

Parte III. Conclusiones

Capítulo 13. Conclusiones y líneas de trabajo futuro

13.1. Conclusiones

Esta tesis ha abarcado cuatro áreas fundamentales de la Ingeniería de Software: especificación formal, reutilización, separación de intereses y aspectos tempranos. Al reunir los elementos de estas cuatro áreas, se ha propuesto una manera de realizar especificaciones reutilizables de requerimientos que contienen esquemas formales en lenguaje Z desde una perspectiva de descomposición que permite encapsular aisladamente cada uno de los intereses junto con toda su información asociada, al mismo tiempo que proporciona medios separados para establecer las relaciones entre ellos mediante reglas de composición; la sintaxis, el uso conjunto de éstas últimas con lenguaje Z (mediante Z-SIIMD) y SPZS, la herramienta de software que implementa este modelo de trabajo, también son contribuciones de esta tesis.

En las siguientes secciones se presentan los resultados obtenidos, aportes y conclusiones identificados por cada una de las principales áreas de conocimiento abordadas en este trabajo.

13.1.1. Especificación formal en lenguaje Z

Para esta tesis se eligió trabajar con lenguaje Z por la madurez que ha alcanzado dentro de las notaciones formales, por ser un estándar internacional, por haber sido utilizado ampliamente en proyectos reales y por contar con una buena cantidad de fuentes bibliográficas de referencia.

La unidad de especificación clave para trabajar conceptualmente en Lenguaje Z es el esquema. Los esquemas pueden ser combinados e incluidos unos dentro de otros. Sin embargo, en Z no existen reglas de composición que permitan definir el impacto de un esquema sobre otro. El presente trabajo propone el uso de reglas de composición aisladas de la especificación formal, para indicar las relaciones de proyección entre los requerimientos. Las reglas proporcionan los elementos necesarios para hacer referencia a los requerimientos involucrados y realizar las adecuaciones necesarias según sea la relación especificada en las reglas; el objetivo es que posteriormente al proceso de integración, se obtenga un elemento final compuesto que muestre los efectos de la regla de composición.

El formato, la sintaxis y los elementos contenidos en las reglas de composición de Z-SIIMD son propuesta de este trabajo.

13.1.2. Reutilización

La separación multidimensional de intereses facilitó la reutilización de los elementos permitiendo que tuvieran la menor cantidad posible de dependencias entre ellos y que permanecieran como entidades separadas en la mayoría de los escenarios.

Se emplearon repositorios para almacenar los elementos que estaban planeados para reutilizarse; los repositorios completos están siempre disponibles para cada uno de los proyectos de especificación que se realicen.

El acceso a los repositorios se obtiene mediante dos procesos denominados "clasificar" e "instanciar". Ambos términos fueron adoptados por la influencia de esos mismos conceptos en el paradigma de las líneas de productos, un enfoque conocido y probado de reutilización en entornos donde se desarrollan diferentes productos de software dentro de un mismo dominio.

El proceso de clasificación es el encargado de enviar elementos al repositorio, mientras que el proceso de instanciación incorpora elementos del repositorio en algún proyecto. En los repositorios se pueden almacenar requerimientos, intereses y reglas de composición.

Con los resultados obtenidos del uso de SPZS se comprobó que la reutilización de elementos con especificaciones formales utilizando el modelo propuesto por Z-SIIMD resulta viable siempre y cuando exista también una manera de adaptar los elementos al momento de incorporarlos (instanciarlos) dentro de un proyecto. Esta adaptación permite sobrepasar las dificultades que resultan de incorporar entidades externas a un nuevo proyecto. Entre las principales dificultades se encuentran los conflictos de datos de identificación entre los elementos a instanciar y los elementos que ya existen en el proyecto en ese momento.

13.1.3. Separación de intereses y aspectos tempranos

En este trabajo se esbozó un modelo de representación para intereses, requerimientos y reglas de composición que incluye datos de identificación, localización, y descripción de los elementos utilizando lenguaje natural y formal en lenguaje Z. Con el modelo propuesto se demostró que es factible mantener los intereses especificados separados entre sí, y aislados de las relaciones que describen su proyección. La organización de los intereses se ve beneficiada, al mismo tiempo que las dependencias entre ellos se minimizan.

Para la separación de intereses se utilizó un enfoque influenciado por el modelo multidimensional de separación de intereses propuesto en (TARR, Peri and Sutton, Jr., Stanley M., 1999). El hecho de que en esta tesis se persiguiera también la reutilización de elementos influenció altamente esta decisión.

El modelo multidimensional propuesto por Tarr y Sutton no es presentado a detalle por los autores ni tampoco adaptado a alguna actividad específica del ciclo de vida en el trabajo original. En el presente trabajo se adaptó ese modelo para la etapa de especificación de requerimientos y se acotaron sus conceptos para trabajar en un prototipo de software que, en una etapa inicial, permitiera comprobar la validez de Z-SIIMD. Por los primeros resultados obtenidos se encontró que trabajar con una separación multidimensional de intereses realmente facilita la reutilización y proporciona una mayor facilidad de configuración en el sistema especificado. También se encontró que después de las experiencias con la versión inicial del prototipo, resulta factible pensar en incorporar otros elementos del modelo como el manejo de dimensiones simultáneas.

13.1.4. Implementación

El software realizado en esta tesis (SPZS) permitió mostrar que el uso de las reglas de composición y el proceso de integración correspondiente producían los resultados esperados; en general reveló que el trabajo se encontraba bien encaminado.

El algoritmo de composición guiado por reglas también es una propuesta de este trabajo de tesis, y fue el primer proceso en ser probado. En las etapas iniciales del desarrollo de SPZS, no existía entorno gráfico ni interacción con el usuario; las pruebas se realizaban desde el mismo código fuente del programa, utilizando las clases creadas para el proyecto.

La versión actual del software es un prototipo evolutivo realizado en *C#* bajo el paradigma de orientación a objetos que utiliza *Windows Forms* para mostrar la interfaz de usuario y *db4Objects* para manejar los datos persistentes como los repositorios y los proyectos. Se decidió utilizar *Windows Forms* para proporcionar más flexibilidad en la interacción con el usuario, y se decidió usar *db4Objects* por la versatilidad que proporciona al trabajar con objetos persistentes, además de ser fácil de utilizar, configurar y ser de código abierto.

13.2. Líneas de trabajo futuro

En este apartado se presentan las líneas de trabajo futuro divididas en dos categorías: “continuación del trabajo”, que agrupa aquellas líneas relacionadas directamente con esta tesis porque completan o complementan características susceptibles de mejora inmediata; y el apartado “líneas complementarias” que engloba las particularidades que, aunque sin duda robustecen la presente tesis, pueden realizarse posteriormente o en forma paralela al resto de características identificadas como especialmente primordiales.

13.2.1. Continuación del trabajo

Los principales puntos de continuación del presente trabajo de tesis se enumeran a continuación:

- Enriquecer los modelos de los elementos propuestos en el apartado 7.2 (requerimientos, intereses y reglas de composición) para aumentar el número de características asociadas a ellos. El grado de detalle incorporado estará en función del alcance que se considere como meta para trabajar con Z-SIIMD y con SPZS. Así, se pueden agregar elementos relevantes a diferentes objetivos y actividades; por ejemplo, algunas características a añadir podrían ser: fuente del requerimiento, prioridad, estado actual en el desarrollo del proyecto, datos de prueba, información de certificación, datos de revisión y detalles de compatibilidad.

- Estudiar la posibilidad de agregar nuevos elementos a las reglas de composición para aumentar las opciones de relación entre los requerimientos. Se debe analizar con detalle el impacto y la utilidad de los nuevos elementos tomando como fundamento las experiencias de las especificaciones realizadas con la notación del presente trabajo de tesis.

- Buscar la mejor manera de agregar en Z-SIIMD el uso de esquemas parametrizados para aumentar los medios de reutilización. La inclusión de parámetros en los esquemas hace posible la implementación del concepto de esquemas genéricos permitidos en Lenguaje Z estándar; de esta manera las especificaciones pueden ser las mismas para diferentes tipos de datos.

- Analizar el impacto de adoptar una instanciación modificada o más amplia del modelo multidimensional en el presente enfoque de trabajo sin aumentar la complejidad para el desarrollador. Conforme se vayan incluyendo más etapas del ciclo de vida en el rango de acción de SPZS, surgirán nuevas maneras de enriquecer el modelo propuesto con Z-SIIMD, las cuales pueden modificar la forma actual de trabajo al permitir enfrentar mejor la complejidad en el proceso completo de desarrollo. Por otra parte, si bien es cierto que el modelo multidimensional es muy flexible y ofrece muchas ventajas, puede agregar complejidad al proceso de especificación si no es bien adaptado y delimitado. Ante cualquier cambio propuesto, se debe buscar un equilibrio entre funcionalidad y complejidad.

- Mejorar la herramienta de software SPZS aumentando las características identificadas y explicadas en el apartado 10.4, las cuales se pueden resumir en: incorporar más elementos del modelo multidimensional de separación de intereses, apegar la captura y representación de las especificaciones formales a una sintaxis validada, incorporar más elementos al buscar y categorizar elementos, incrementar el número de formatos en el que se presenta el modelo completo y fortalecer los mecanismos de trazabilidad entre las diferentes etapas del ciclo de vida.

13.2.2. Líneas complementarias

Las principales líneas complementarias de trabajo relacionadas con el presente trabajo de tesis son:

- Extender SPZS para soportar varios lenguajes formales en las especificaciones. Actualmente esta versión del prototipo está pensada para trabajar solo con Lenguaje Z; sin embargo resultaría conveniente ampliarla mediante el uso de componentes y librerías para seleccionar el lenguaje con el que se desea trabajar. Las reglas de composición pueden conservar su sintaxis o pueden ser ampliadas según el lenguaje seleccionado.

- Implementar nuevos algoritmos de búsqueda, clasificación y recuperación en el manejo de los repositorios. Actualmente el proceso de reutilización se basa en la gestión de un repositorio básico con inserciones y recuperaciones genéricas; sin embargo, se puede robustecer con el uso de algoritmos más sofisticados de búsqueda, clasificación y recuperación que tomen en cuenta la sintaxis, la semántica o el contexto en el que se encuentran los elementos. Los nuevos algoritmos pueden ofrecer también la capacidad de trabajar con diferentes versiones documentadas de un mismo elemento que se encuentra en el repositorio. Las diferentes versiones podrían surgir como producto de la evolución o como variantes particulares de adaptación a un contexto particular.

- Diseñar e implementar un algoritmo que permita la construcción inteligente, semi-automatizada y guiada de reglas de composición a partir de un análisis de compatibilidad (tipos de datos, entradas y salidas) entre los requerimientos de la especificación.

- Medir el impacto benéfico de la separación de intereses propuesto por este trabajo en la reutilización, trazabilidad, y facilidad de mantenimiento de un proyecto real, comparando los resultados con los obtenidos al aplicar un proceso tradicional completamente orientado a objetos realizado para el mismo proyecto.

- Incorporar a Z-SIIMD y SPZS soporte para técnicas de diseño arquitectural y diseño detallado que permitan establecer vínculos precisos entre los elementos de especificación y los de diseño. La expansión del rango de acción de SPZS hacia el resto de las actividades del ciclo de vida puede comenzar de esta manera, y puede continuar su evolución hasta convertirse en una útil y completa herramienta CASE orientada a aspectos.

Apéndice: Bases de Lenguaje Z

Capítulo 14.
Bases del lenguaje Z

14.1. ¿Qué es el lenguaje Z?

El lenguaje Z es una notación formal basada en estados desarrollada en la década de los ochenta por el grupo "Programming Research Group" de la Universidad de Oxford en combinación con otros aliados industriales; desde el año 2002 es considerado un estándar internacional (ISO/IEC-13568, 2002).

Fundamentalmente se utiliza para realizar especificaciones de operaciones en una forma declarativa basada en entradas, salidas y condiciones. Como todo lenguaje, Z tiene una gramática propia, la cual está basada en las matemáticas clásicas, es decir, en lógica de dos valores y teoría de conjuntos.

Como el lenguaje Z se apega al uso de las matemáticas, es posible razonar acerca de las especificaciones y realizar pruebas con ellas para asegurarse de que algunas condiciones se cumplan en consecuencia. Esto es particularmente relevante en sistemas donde la seguridad tiene una importancia crítica.

Realizar una especificación de requerimientos de un sistema en lenguaje Z implica modelar los estados de ese sistema, usando entre otras cosas: esquemas con declaraciones de variables y conjuntos de algún tipo de dato específico, relaciones entre esas variables, e invariantes (condiciones, restricciones).

En los siguientes apartados se expondrán los fundamentos de los elementos más relevantes para realizar especificaciones formales en Z.

14.2. Tipos

En Z, todos los objetos tienen un tipo asociado. El tipo de los objetos determina la compatibilidad de las operaciones que se realizan con ellos. De esta manera, solo son válidas las operaciones con objetos de tipos compatibles.

El tipo de un objeto es el conjunto máximo o más inclusivo al cual pertenece (JACKY, Jonathan, 2008). El especificador puede definir sus propios tipos de acuerdo a las necesidades de su proyecto, ya que solo el tipo entero, denotado por el símbolo $\mathbb{Z}$, que incluye los números enteros negativos, positivos y el cero, se encuentra nativamente definido y listo para usarse en Z.

Dependiendo de la manera de describir los valores aceptados para un tipo, éste puede ser básico o libre. En un tipo libre, la definición contiene explícitamente en un listado los posibles elementos que lo conforman. Se utiliza esta declaración cuando los elementos del tipo son pocos y pueden ser enumerados uno por uno. Primero se coloca el nombre del tipo seguido de ::= y después, los elementos separados por el carácter barra (|).

Por ejemplo, para crear un nuevo tipo que describa los diferentes grupos de usuarios que tiene un sistema, se realiza la siguiente definición:

TIPOUSUARIO ::= *administrador* | *capturista* | *reporteador*

Por otra parte, los tipos básicos se utilizan cuando se trabaja con conjuntos muy grandes, de los cuales se omiten los elementos al momento de la declaración. Para definirlos, se utiliza el nombre del tipo encerrado en paréntesis cuadrados ([ ]).

Por ejemplo, para crear un nuevo tipo que describa los diferentes nombres de usuarios registrados en un sistema, se realiza lo siguiente:

[*NOMBREUSUARIO*]

En un mismo párrafo se pueden realizar varias definiciones. En el siguiente ejemplo se crean dos tipos diferentes: uno para denotar el nombre de los servidores en una instalación de red y otro para denotar los nombres de los recursos disponibles:

[*SERVIDOR*, *RECURSO*]

14.3. Conjuntos

Un conjunto es una colección de componentes llamados elementos o miembros. Una característica importante de los conjuntos es que el orden de sus elementos es irrelevante, por lo que no se puede hablar del primer o último miembro. Los elementos de un conjunto se separan por comas y se delimitan con llaves { }. Por ejemplo {A, B, C} es un conjunto de 3 elementos: A, B, y C. Un conjunto vacío es aquel que no tiene elementos. Se representa con el símbolo $\varnothing$ o bien con llaves solas { }.

En Z, todos los conjuntos tienen un tipo determinado, y todos los elementos de un mismo conjunto son de igual tipo. Las operaciones que se pueden realizar entre dos o más conjuntos son permitidas si los conjuntos involucrados son del mismo tipo.

Todos los tipos son conjuntos, pero no todos los conjuntos son tipos. Por ejemplo, $\mathbb{N}$ es un conjunto que contiene los números naturales {0, 1, 2, 3,…}, pero no es un tipo. El tipo de $\mathbb{N}$ en realidad es $\mathbb{Z}$ pues cada número natural es también un entero. $\mathbb{Z}$ representa el conjunto de todos los enteros {…-3, -2, -1, 0, 1, 2, 3…}, de esta manera se puede ver claramente que los elementos de $\mathbb{N}$ están contenidos en el conjunto especificado por $\mathbb{Z}$. (JACKY, Jonathan, 2008).

14.4. Operadores de conjuntos

Estos operadores sirven para realizar operaciones con los conjuntos. Algunos de los operadores más comunes son: equivalencia, no equivalencia, pertenencia, cardinalidad, subconjunto, unión, intersección, y diferencia. Todos ellos se encuentran explicados en el Cuadro 14.1.

Nombre del operador	Símbolo usado en Z	Propósito	Ejemplo
Equivalencia	$=$	Determinar si dos conjuntos contienen exactamente los mismos elementos.	$\{1,2,3\} = \{3,2,1\}$
No equivalencia	$\neq$	Determinar si dos conjuntos contienen elementos diferentes.	$\{1,2\} \neq \{1,2,3\}$
Pertenencia	$\in$	Determinar si un elemento es miembro de un conjunto.	$1 \in \{1,2,3\}$
No pertenencia	$\notin$	Determinar si un elemento no es miembro de un conjunto.	$3 \notin \{1,2\}$
Tamaño o cardinalidad	#	Determinar el número de elementos en un conjunto.	$\#\{1,2,3\} = 3$
Subconjunto	$\subseteq$	Determinar si un conjunto está contenido o es igual a otro.	$\{1,2\} \subseteq \{1,2,3\}$
Unión	$\cup$	Obtener un conjunto que contiene los elementos que están en el primer conjunto o en el segundo.	$\{1,2,3\} \cup \{1,4,5\}=\{1,2,3,4,5\}$
Intersección	$\cap$	Obtener un conjunto que contiene los elementos que están en el primer conjunto y también se encuentran en el segundo.	$\{1,2,3\} \cap \{1,4,5\} = \{ 1 \}$
Diferencia	$\setminus$	Obtener un conjunto que contiene todos los elementos del primer conjunto que no están en el segundo conjunto.	$\{1,2,3\} \setminus \{3,4\} = \{1, 2\}$

Cuadro 14.1 Operadores de conjuntos

14.5. Declaraciones y variables

Los objetos deben ser declarados antes de utilizarse. Una declaración determina el nombre de un objeto y el conjunto al cual pertenece, que no necesariamente tiene que ser un tipo. Se pueden declarar variables para denotar un solo elemento o un conjunto de elementos. En

ambos casos se puede declarar una sola variable por renglón, o varias al mismo tiempo. Para declarar varias variables en un mismo renglón basta con separarlas con comas. Los siguientes son ejemplos de declaraciones de variables que se refieren a un solo elemento:

$nu : NOMBREUSUARIO$
$u1, u2 : TIPOUSUARIO$
$s, t, v : SERVIDOR$

Es así como *nu* representa un nombre de usuario; *u1* y *u2* representan cada uno un tipo de usuario; y *s, t, v* son tres servidores cada uno independiente del otro.

Además de los tipos, en una declaración también pueden aparecer directamente rangos de números y conjuntos. Por ejemplo, las siguientes declaraciones son válidas para indicar un año entre 1970 y 1990, un mes ya sea junio o julio y un número de día entre 1 y 10.

$a : 1970..1990$
$m : \{junio, julio\}$
$d : 1..10$

De la misma manera se pueden declarar variables que denoten conjuntos de elementos, para lo cual se utiliza el símbolo $\mathbb{P}$ que es una abreviatura para "Power Set" y se refiere al conjunto de todos los subconjuntos. Por ejemplo, el conjunto de servidores instalados en un centro de investigación podría estar denotado por:

$servs : \mathbb{P}\ SERVIDOR$

Esta declaración indica que *servs* es un conjunto de elementos de tipo *SERVIDOR*.

14.6. Variables globales y definiciones axiomáticas

Una definición axiomática es aquella que introduce declaraciones de ámbito global; éstas pueden ser utilizadas en cualquier parte de la especificación después de ser definidas la primera vez. Las declaraciones globales no cambian su valor una vez que éste ha sido asignado.

La representación de una definición axiomática es una caja abierta con divisiones para declaraciones y predicados. Los predicados restringen los valores que pueden tomar las variables definidas. La restricción puede ser para un solo valor o para un conjunto de valores.

En el siguiente ejemplo se define el número máximo de aplicaciones cliente que pueden estar conectadas simultáneamente en un sistema informático:

$$\begin{array}{|l} maxClientes : \mathbb{N} \\ \hline maxClientes \leqslant 1024 \end{array}$$

En este ejemplo se ha especificado que el número máximo de clientes siempre debe ser menor o igual a 1024. Posteriormente se puede elegir un número particular (siempre menor o igual a 1024) según la configuración que se desee dar al sistema informático, sin embargo, una vez elegido, el valor se mantiene sin cambio.

14.7. Constantes y definiciones abreviadas

Una constante toma siempre un único valor, el cual no se modifica a lo largo de toda la especificación. Para realizar la declaración de una constante se puede utilizar una definición axiomática o una definición abreviada.

En el siguiente ejemplo se define una constante para la tasa de impuestos a cobrar en un sistema contable utilizando una definición axiomática:

$$\begin{array}{|l} impuesto : N \\ \hline impuesto = 16 \end{array}$$

Por otra parte, una definición abreviada crea un sinónimo para conjuntos y tipos. El sinónimo creado se puede utilizar en toda la especificación. Para crear una definición abreviada se utilizan dos símbolos de igual (==).

A continuación se define la misma constante para la tasa de impuestos del ejemplo anterior utilizando ahora una definición abreviada:

$$impuesto == 16$$

Otros ejemplos de definiciones abreviadas se muestran a continuación:

$$LASTIP == 114 \mathrel{..} 127$$
$$GRUPOUSUARIOS == \{ A, B, C, D, E \}$$

Al utilizar definiciones abreviadas no es necesario realizar la declaración del tipo del sinónimo, ya que solo representa otra manera de nombrar a un elemento.

14.8. Tuplas y relaciones

Cuando es necesario definir un elemento que contenga en sí mismo varios elementos de diferentes tipos (por ejemplo un registro de un archivo o una base de datos), se requiere trabajar con tuplas.

El tipo de una tupla es el producto cartesiano de cada uno de los tipos de los elementos que contiene. Por ejemplo, suponer que se desea modelar el conjunto de datos asociado a un usuario de un sistema de cómputo; los elementos a considerar serían: número de usuario, nombre, tipo de usuario. Primero se deben definir los tipos de los elementos individuales:

$NUMEROUSUARIO == \mathbb{N}$
$[NOMBRE]$
$TIPOUSUARIO ::= administrador \mid capturista \mid mantenimiento$

Luego, se define el tipo de la tupla, como el producto cartesiano de los tipos involucrados:

$USUARIO == NUMEROUSUARIO \times NOMBRE \times TIPOUSUARIO$

Y finalmente ya se puede declarar una variable para representar un usuario:

$unUsuario : USUARIO$

Si lo que se deseara representar fuera un conjunto de usuarios (un conjunto de tuplas), se necesita una relación. Para declararla se procede de la siguiente manera:

$losUsuarios : \mathbb{P}\ USUARIO$

Una asignación de valores típica para una tupla sería:

$unUsuario = (5294, Charles\ Brown, administrador)$

Y una asignación de valores típica para una relación sería:

$losUsuarios = \{ (5294, Charles\ Brown, administrador), (5296, Martin\ Jules, capturista) \}$

14.9. Pares

Si una tupla tiene solamente dos elementos, se le llama par. Por ejemplo, supóngase que la siguiente tupla contiene la información del nombre de un servidor y el nombre de la ciudad donde se encuentra físicamente instalado:

$(alfa3, Monterrey)$

Alternativamente a esta notación, los pares pueden utilizar el símbolo del *maplet* ($\mapsto$) de la siguiente manera:

$alfa3 \mapsto Monterrey$

Para extraer el primer y segundo elemento de un par se utilizan los operadores *first* y *second* respectivamente. Ejemplo:

$first\ (alfa3, Monterrey) = alfa3$
$second(alfa3, Monterrey) = Monterrey$

14.10. Relaciones binarias

Una relación binaria es un conjunto de pares. Para definir un conjunto de pares denominado *ubicacionServidores* para el mismo ejemplo anterior, y primero definiendo los tipos correspondientes se tiene:

$[SERVIDOR]$
$[CIUDAD]$
$UBICACION == SERVIDOR \times CIUDAD$
$ubicacionServidores : \mathbb{P}\ UBICACION$

Para el caso particular de una relación binaria existe una notación alternativa al producto cartesiano. Se trata del símbolo de flecha con doble punta ($\longleftrightarrow$) de tal manera que:

$UBICACION == SERVIDOR \longleftrightarrow CIUDAD$

es equivalente a:

$UBICACION == SERVIDOR \times CIUDAD$

En una relación binaria, para obtener el conjunto de todos los primeros elementos se utiliza la función *dom* que es una abreviatura de dominio; para obtener el conjunto de todos los segundos elementos se utiliza la función *ran* que es una abreviatura de rango. Por ejemplo, considerar:

ubicacionServidores = { (*alfa*3, *Monterrey*), (*beta*1, *Guadalajara*), (*gamma*2, *Monterrey*) }

Al aplicar las funciones de dominio y rango sobre *ubicacionServidores* se obtiene el siguiente resultado:

dom(*ubicacionServidores*) = { *alfa*3, *beta*1, *gamma*2 }
ran(*ubicacionServidores*) = { *Monterrey*, *Guadalajara* }

14.11. Operadores para las relaciones binarias

Los operadores para las relaciones binarias son: imagen relacional, restricción del dominio, restricción del rango, anti-restricción del dominio, anti-restricción del rango, sobre-escritura, inversión y composición relacional. Todos ellos se encuentran explicados en el Cuadro 14.2. Para cada uno de los ejemplos mostrados en ese cuadro se ha tomado como punto de partida la relación binaria *ubicacionServidores* tal como se encuentra definida en el apartado 14.10.

Nombre y propósito	**Uso general en Z**	**Ejemplo**
Imagen relacional.- Realizar búsquedas de elementos del rango, proporcionando elementos del dominio.	nombreRelacion ⦇ { elementos del dominio } ⦈ = {elementos del rango}	ubicacionServidores ⦇ {alfa3, beta1 } ⦈ = {*Monterrey, Guadalajara*}
Restricción del dominio.- Realizar búsquedas de tuplas completas que coincidan con elementos del dominio.	{elementos del dominio} ◁ nombreRelacion	{ alfa3, beta1} ◁ ubicacionServidores = { (*alfa*3, *Monterrey*), (*beta*1, *Guadalajara*) }
Restricción del rango.- Realizar búsquedas de tuplas completas proporcionando elementos del rango.	nombreRelacion ▷ {elementos del rango }	ubicacionServidores ▷ { Monterrey, Guadalajara } = { (*alfa*3, *Monterrey*), (*beta*1, *Guadalajara*), (*gamma*2, *Monterrey)* }

Cuadro 14.2 Operadores para las relaciones binarias. (Continúa).

Nombre y propósito	Uso general en Z	Ejemplo
Anti-Restricción del dominio.- Realizar búsquedas de tuplas completas que no coincidan con elementos del dominio.	{elementos del dominio} ⩤ nombreRelacion	{alfa3, beta1} ⩤ ubicacionServidores = { (gamma2, Monterrey) }
Anti-Restricción del rango.- Realizar búsquedas de tuplas completas que no coincidan con elementos del rango.	nombreRelacion ⩥ {elementos del rango }	ubicacionServidores ⩥ { Monterrey } = { (beta1, Guadalajara) }
Sobre-Escritura.- Realizar actualizaciones o inserciones de pares.	nombreRelacion ⊕ {tuplas a modificar o agregar}	ubicacionServidores ⊕ {(gamma2,Mérida),(beta2,Guadalajara)} = { (alfa3, Monterrey), (beta1, Guadalajara), (gamma2, Mérida), (beta2, Guadalajara) }
Inversión.- Intercambiar el primer y segundo elemento de cada par.	nombreRelacion~	ubicacionServidores~ = { (*Monterrey, alfa3*), (*Guadalajara, beta1*), (*Monterrey, gamma2*) }
Composición relacional.- Combina dos relaciones en una sola tomando en cuenta los elementos iguales de ambas.	nombreRelacion1 ⨾ nombreRelacion2	Considerar la siguiente relación: ubicacionCiudades = {(Monterrey, Nuevo León), (Guadalajara, Jalisco)} Entonces: ubicacionServidores ⨾ ubicacionCiudades = { (alfa3, Nuevo León), (beta1, Jalisco), (gamma2, Nuevo León) }

Cuadro 14.2. (Continuación) Operadores para las relaciones binarias.

14.12. Funciones

Una función es un caso especial de una relación binaria en donde hay como máximo un valor en el rango por cada valor en el dominio. Cada elemento del dominio de la función puede aparecer únicamente una vez en los pares de la relación; sin embargo, los elementos del rango pueden aparecer varias veces.

Una función puede ser total o parcial. Cuando una función no se aplica universalmente para todos los elementos del conjunto fuente (origen de los elementos del dominio), se dice que la función es parcial y para su representación se utiliza el símbolo ($\rightarrow\!\!\!\!+$). Estos casos son los más comunes de encontrar en situaciones cotidianas. Si se aplica la función a un elemento que no devuelva un resultado, se dice que la función carece de significado para ese caso particular.

Por el contrario, si la función se aplica para todos los elementos del conjunto fuente, se trata de una función total y para denotarla se emplea el símbolo de la flecha completa plana ($\longrightarrow$).

El siguiente ejemplo muestra una función parcial: se desean relacionar las ciudades y los estados a los que pertenecen, para lo cual se definen los tipos de la siguiente manera:

$$CIUDAD ::= NuevoLaredo \mid CiudadVictoria \mid Monterrey \mid Reynosa$$
$$ESTADO ::= Tamaulipas \mid NuevoLeon$$

Para asignar valores se realiza lo siguiente:

$$\begin{array}{|l}
cds : CIUDAD \rightarrow\!\!\!\!+ ESTADO \\
\hline
cds = \{ NuevoLaredo \mapsto Tamaulipas, \\
\qquad CiudadVictoria \mapsto Tamaulipas, \\
\qquad Monterrey \mapsto NuevoLeon \}
\end{array}$$

Para encontrar el elemento correspondiente asociado con otro mediante una función, se procede de la siguiente manera:

Por ejemplo, para saber el estado en el que se encuentra la ciudad NuevoLaredo, se puede escribir:

$$cds(NuevoLaredo) = Tamaulipas$$

O también sin paréntesis:

$$cds\ \ NuevoLaredo = Tamaulipas$$

A esta operación se le denomina Aplicación de la función y es un caso particular del operador de imagen relacional. Por ejemplo, la siguiente definición es equivalente a las dos anteriores:

$$cds\ \ (\!|\ \{NuevoLaredo\}\ |\!) = Tamaulipas$$

Como ésta es una función parcial, no está definida para todos los elementos del dominio. Por ejemplo la expresión:

$cds(\ Reynosa\ )$

no tiene un resultado asociado, por lo tanto, se dice que no tiene significado en este caso particular.

Por otra parte, algunas funciones matemáticas son casos típicos de funciones totales. Por ejemplo, todos los números naturales tienen una raíz cuadrada asociada a ellos. Esto puede definirse como:

$Raiz\_Cuadrada: N \longrightarrow N$

Otro caso especial de funciones son las inyecciones o funciones inyectivas, que asocian cada elemento de su dominio con un diferente elemento de su rango. Las inyecciones representan una relación uno a uno, por lo que son reversibles. Si la inyección es parcial para representarlas se utiliza el símbolo ($\rightarrowtail\!\!\!\!\!+$); si se trata de una inyección total, se usa el símbolo ($\rightarrowtail$).

Por ejemplo, en México, cada persona física tiene un único RFC (Registro Federal de Contribuyentes), el cual tiene que ser gestionado por el interesado ante el Sistema de Administración Tributaria, ahí se le asigna una clave que consiste en cuatro letras según su nombre completo, seis dígitos según su fecha de nacimiento y tres caracteres adicionales. Como se requiere un trámite expreso por parte de cada ciudadano, no todos cuentan con un RFC de esta naturaleza.

Para representar esta relación inyectiva y parcial, e incluir dos pares con datos de muestra arbitrarios, se puede utilizar la siguiente definición:

$$
\begin{array}{|l}
rel: PERSONA \rightarrowtail\!\!\!\!\!+ RFC \\
\hline
rel = \{\ ramiro \mapsto rofr780210CW2, \\
\qquad alicia \mapsto fafa750105RG5\ \ \}
\end{array}
$$

14.13. Secuencias

Una secuencia es un conjunto de elementos cuyo orden es importante, por lo que se puede acceder al valor de cualquiera de ellos si se conoce su posición; además con una secuencia es posible realizar operaciones como obtener el primero, o el último elemento.

Una secuencia se declara usando la palabra *seq* y sus elementos se escriben separados por comas y delimitados por los caracteres ⟨ ⟩. Por ejemplo, para definir los grupos de permisos a los que pertenecen los usuarios de un sistema, se declara primero un nuevo tipo:

[*GRUPOPERMISO*]

Posteriormente, si se desea representar ordenadamente el grado de restricción de los usuarios que se encuentran conectados al sistema en un momento determinado, se debe declarar una secuencia de la siguiente manera:

grupo : *seq GRUPOPERMISO*

Una secuencia vacía es aquella que no tiene ningún elemento y se representa con los símbolos ⟨ ⟩. Si *grupo* estuviera vacía, se representaría de la siguiente manera:

grupo = ⟨ ⟩

Y si se deseara asignar inicialmente elementos a la secuencia *grupo*, se debe realizar de la siguiente manera:

grupo = ⟨ *B*, *C* ⟩

El número de elementos que se encuentran en una secuencia se denota con el símbolo #. Por ejemplo, para representar el número de elementos en la secuencia grupo, se escribe lo siguiente:

#*grupo* = 2

14.14. Operadores de secuencias

Los operadores de secuencias son: selección, cabeza, cola, último, frente, concatenación, filtrado, reversa y rango. Todos ellos se encuentran explicados en el Cuadro 14.3. Para cada uno de los ejemplos mostrados, se ha considerado la siguiente secuencia:

grupo = ⟨ *B*, *C*, *D* ⟩

Nombre del operador	Símbolo usado en Z	Propósito	Ejemplo
Selección	*nombresecuencia posición*	Obtener el elemento que se encuentra en una determinada posición.	grupo 2 = C
Cabeza	Head	Obtener el primer elemento.	head grupo = B
Cola	Tail	Obtener la secuencia sin el primer elemento.	tail grupo = ⟨ C, D ⟩
Ultimo	Last	Obtener el último elemento.	last grupo = D
Frente	Front	Obtener la secuencia sin el último elemento	front grupo = ⟨ B, C ⟩
Concatenación	⌢	Concatenar dos secuencias.	grupo ⌢ ⟨ E ⟩ = ⟨ B, C, D, E ⟩
Filtrado	↾	Filtrar los elementos de una secuencia que coincidan con un conjunto dado. El resultado es otra secuencia.	grupo ↾ { C, D, E } = ⟨ C, D ⟩
Reversa	Rev	Invertir los elementos.	rev grupo = ⟨ D, C, B ⟩
Rango	Ran	Obtener un conjunto con los valores de la secuencia.	ran grupo = { B, C, D }

Cuadro 14.3 Operadores de secuencias.

14.15. Conectivas Lógicas

Las conectivas lógicas son necesarias para crear expresiones complejas, unidas a partir de otras más simples. Las conectivas más comunes son: conjunción, disyunción, negación, equivalencia, e implicación, todas ellas se encuentran explicadas en el Cuadro 14.4.

Para cada uno de los ejemplos en el cuadro se consideran las siguientes especificaciones relacionadas con un sistema de comunicación cliente-servidor. Los tipos utilizados para denotar servidores y clientes son:

[*SERVIDOR*]
[*CLIENTE*]

Para las declaraciones se han utilizando nombres representativos, de tal manera que, por ejemplo *PCcliente* es un equipo personal que funciona como cliente y *ClientesBloqueados* es el conjunto de todos los equipos cliente que se encuentran bloqueados. Las declaraciones utilizadas son:

numeroClientesConectados : $\mathbb{Z}$
PCcliente : *CLIENTE*
unServidor : *SERVIDOR*
ClientesBloqueados : $\mathbb{P}$ *CLIENTE*
ServidoresDisponibles : $\mathbb{P}$*SERVIDOR*
ServidoresApagados : $\mathbb{P}$ *SERVIDOR*
ServidoresPrendidos: $\mathbb{P}$ *SERVIDOR*

Nombre de la conectiva	Símbolo usado en Z	Tabla de verdad	Ejemplo
Conjunción	$\wedge$	p q p ∧ q F F F F V F V F F V V V	numeroClientesConectados > 0 ∧ numeroClientesConectados ⩽ 100
Disyunción	$\vee$	p q p ∨ q F F F F V V V F V V V V	#ServidoresDisponibles = 0 ∨ PCcliente ∈ ClientesBloqueados
Negación	$\neg$	p ¬ p F V V F	¬ (PCcliente ∈ ClientesBloqueados)
Equivalencia	$\Leftrightarrow$	p q p ⇔ q F F V F V F V F F V V V	unServidor ∉ ServidoresApagados ⇔ unServidor ∈ ServidoresPrendidos
Implicación	$\Rightarrow$	p q p ⇒ q F F V F V V V F F V V V	numeroClientesConectados ⩽ 100 ⇒ #ServidoresDisponibles > 0

Cuadro 14.4 Conectivas lógicas.

14.16. Cuantificadores

Un cuantificador permite realizar una o varias restricciones para uno o más elementos, utilizando una sola expresión que abarca todas las posibilidades sin tener que escribir todos los elementos afectados uno por uno. Los cuantificadores introducen variables locales a la situación particular que expresan, por lo que esas variables no son descriptivas del estado global del sistema, y carecen de significado fuera del contexto del cuantificador.

El cuantificador universal ($\forall$) es una generalización de la conjunción ($\wedge$) y tiene la forma:

$$\forall \textit{ declaración} \bullet \textit{predicado}$$

El punto negro grueso entre la declaración y el predicado es un delimitador. La expresión indica que el *predicado* es verdadero *para todos* los valores admitidos en la *declaración*.

Por ejemplo, la siguiente expresión indica que para todo número natural, se cumple la condición de que el número es mayor o igual a cero.

$$\forall x : \mathbb{N} \bullet x \geqslant 0$$

y su significado se puede entender mejor si se plantea la misma expresión cuantificada como varias conjunciones:

$$0 \geqslant 0 \wedge 1 \geqslant 0 \wedge 2 \geqslant 0 \wedge 3 \geqslant 0 \wedge 4 \geqslant 0 \wedge 5 \geqslant 0 \wedge \ldots$$

En lenguaje Z también se dispone de un cuantificador existencial ($\exists$) que es una generalización de la disyunción ($\vee$) y tiene una forma similar al cuantificador universal. El cuantificador existencial expresa que existe uno o varios valores introducidos en la declaración para las cuales el predicado es verdadero.

Por ejemplo, la siguiente expresión indica que existen (uno o varios) números enteros que son mayores que cero:

$$\exists x : \mathbb{Z} \bullet x > 0$$

El significado del cuantificador se puede entender mejor si se examinan con detalle algunos de los valores implicados en su declaración:

$$\ldots -2 > 0 \vee -1 > 0 \vee 0 > 0 \vee 1 > 0 \vee 2 > 0 \ldots$$

Existe también el cuantificador único ($\exists_1$) similar al existencial, pero más restrictivo: implica que existe un solo valor que hace verdadero el predicado asociado.

Por ejemplo, para expresar que en todo el rango de números enteros existe solo uno que es igual a cero (que es el mismo cero), se indica lo siguiente:

$$\exists_1 x : \mathbb{Z} \bullet x = 0$$

14.17. Esquemas

Un esquema es una estructura básica que Z provee para llevar a cabo especificaciones de una manera estructurada. Los esquemas se representan con una forma de rectángulo abierto por el lado derecho. El rectángulo tiene una división intermedia que da lugar a dos espacios: uno para las declaraciones y otro para los predicados. El nombre del esquema se coloca en el borde superior del rectángulo.

La forma general de un esquema es la siguiente:

NombreDelEsquema

Declaracion 1
Declaracion 2
..
Declaracion n

Predicado 1
Predicado 2
..
Predicado n

Aunque se omite por convención, en realidad cada línea de declaraciones termina con un punto y coma (;). De la misma manera, aún cuando no se expresa explícitamente, cada línea de predicados se relaciona con el resto con el símbolo lógico de conjunción ($\wedge$). Entonces, el mismo esquema anterior es equivalente a:

NombreDelEsquema

Declaracion 1;
Declaracion 2;
..
Declaracion n

Predicado 1 ∧
Predicado 2 ∧
..
Predicado n

Cuando un esquema es pequeño, puede ser más conveniente formarlo de manera horizontal. En esta alternativa se utiliza el símbolo (≙) para separar el identificador de la definición del esquema. Se utiliza el símbolo de barra (|) para separar las declaraciones de los predicados, y el esquema completo se encierra entre corchetes ([ ]). A continuación se muestra la forma general de esta alternativa para crear esquemas:

NombreEsquema ≙ [*Declaraciones* | *Predicados*]

Las declaraciones que se hacen en los esquemas introducen variables de alcance local; esto quiere decir que no es posible nombrar estas variables fuera de ese esquema. Sin embargo, si esto resultara deseable, se puede utilizar una inclusión, la cual se explicará más adelante en este mismo apartado.

Las variables introducidas en los esquemas pueden ser de entrada, de salida o de trabajo. Por convención las variables de entrada terminan su nombre con un símbolo de interrogación (?), las de salida terminan con un símbolo de exclamación (!), y las de trabajo no utilizan ningún símbolo al final. A continuación se muestra un ejemplo de un esquema que utiliza variables de entrada y salida.

CalcularCubo

numero? : ℤ
cubo! : ℤ

---

cubo! = *numero*? * *numero*? * *numero*?

El esquema representa la operación de calcular el cubo del número entero denotado por la variable *numero?*. El resultado se entrega en la variable *cubo!*. Nótese la representación de *numero?* y *cubo!* como variables de entrada y salida respectivamente.

Los esquemas definen el estado de los objetos. Por ejemplo, el siguiente esquema modela un conjunto de usuarios accediendo simultáneamente a una base de datos, y agrega la restricción de que el total de ellos nunca puede ser mayor de 200:

$[USER]$

UsuariosConectados

$u : \mathbb{P}\ USER$

$\#u \leqslant 200$

Una operación que cambie el estado de *UsuariosConectados* permite distinguir los valores del esquema en dos momentos diferentes: antes y después de que la operación se haya llevado a cabo.

En Z, el estado o valor posterior a una operación se denota con el símbolo de prima (′) al final del nombre de un elemento. El mismo nombre sin el símbolo de prima representa el mismo estado pero previamente a la operación en cuestión.

Siguiendo con el ejemplo anterior, el esquema *UsuariosConectados'* representaría el estado después de una operación que, por ejemplo, agregue un nuevo usuario en el registro del sistema, y estaría definido de la siguiente manera:

UsuariosConectados′

$u' : \mathbb{P}\ USER$

$\#u' \leqslant 200$

Para denotar ambos momentos de las variables: antes y después de una operación que cambie el estado, se utiliza el símbolo Delta (Δ). En el ejemplo anterior, *ΔUsuariosConectados* quedaría definido de la siguiente manera:

ΔUsuariosConectados

UsuariosConectados

UsuariosConectados′

Es importante hacer notar que *ΔUsuariosConectados* es un nuevo esquema que se ha representado como la combinación de dos esquemas diferentes que se han incluido totalmente usando solo sus nombres.

Es justamente así como Z permite la inclusión de unos esquemas dentro de otros. La inclusión consiste en incorporar el nombre de un esquema H en la sección de declaraciones

de otro esquema I; de esta manera, todas las variables definidas en H se pueden acceder también desde I. El contenido de un esquema con inclusión puede ser expandido con sus respectivos contenidos de la siguiente manera:

$$
\begin{array}{|l}
\hline
\Delta UsuariosConectados \\
u, u' : \mathbb{P}\ USER \\
\hline
\#u \leqslant 200 \\
\#u' \leqslant 200 \\
\hline
\end{array}
$$

Una operación que cambia el estado de un elemento definido en un esquema H, se identifica porque contiene el nombre del esquema precedido del símbolo delta (ΔH). Ejemplos de operaciones que cambian el estado son inserciones, actualizaciones, y eliminaciones de elementos.

Una operación que no cambia el estado de un elemento definido en un esquema H, se identifica porque contiene el nombre del esquema precedido por el símbolo xi (Ξ); en este caso, ΞH. Las consultas son operaciones típicas que no cambian el estado de los elementos.

Para ejemplificar estos casos, ahora se va a crear un esquema llamado *AgregarUsuario* que tome un nuevo usuario que ha accedido a la base de datos y lo agregue en el conjunto de usuarios conectados registrados. Para la especificación, se debe tener acceso al objeto *u* definido en *UsuariosConectados*; sin embargo, esto no es posible de hacer directamente pues las declaraciones son locales para cada esquema, es por eso que se va a utilizar la inclusión de *UsuariosConectados* dentro de *AgregarUsuario*. Sin embargo, al agregar un usuario, el estado de *UsuariosConectados* cambiará, pues tendrá un usuario más, por lo que el esquema que se incluye será Δ*UsuariosConectados* de la siguiente manera:

$$
\begin{array}{|l}
\hline
AgregarUsuario \\
\Delta UsuariosConectados \\
nuevo? : USER \\
\hline
u' = u \cup \{ nuevo? \} \\
\hline
\end{array}
$$

Con este mismo ejemplo, ahora se va a crear un esquema llamado *ConsultarUsuarios* para obtener los usuarios conectados en un momento determinado. La consulta no va a generar ningún cambio en los usuarios ya registrados, por lo que se incluye el nombre de *ΞUsuariosConectados* dentro del nuevo esquema.

$$
\begin{array}{|l}
\hline
\textit{ConsultarUsuarios} \\
\Xi UsuariosConectados \\
resultado! : \mathbb{P}\ USER \\
\hline
resultado! = u \\
\hline
\end{array}
$$

14.17.1. Cálculo de esquemas

El cálculo de esquemas permite combinar esquemas independientes para formar otros más complejos. Comúnmente se utilizan los operadores lógicos conjunción ($\wedge$) y disyunción ($\vee$) para llevar a cabo la unión. Una situación típica que se tomará como ejemplo es el aislar diferentes casos para un mismo escenario y al final, combinarlos todos, dando lugar a un solo caso de uso pero con un abanico de posibilidades alternas.

Supóngase que se desea modelar una operación de retiro de fondos de una cuenta bancaria. Primero, se define el estado del sistema. Para fines de simplicidad, únicamente se incluirá el saldo de la cuenta, que debe ser siempre mayor o igual a cero:

$$
\begin{array}{|l}
\hline
\textit{CuentaBancaria} \\
saldo : \mathbb{Z} \\
\hline
saldo \geqslant 0 \\
\hline
\end{array}
$$

Luego el estado inicial coloca el saldo de la cuenta a cero:

$$
\begin{array}{|l}
\hline
\textit{InitCuentaBancaria} \\
\Delta CuentaBancaria \\
\hline
saldo' = 0 \\
\hline
\end{array}
$$

Ahora, para realizar un retiro de la cuenta puede haber varias situaciones, para este ejemplo se van a considerar tres: que la cantidad a retirar esté disponible en la cuenta en ese momento, que haya un saldo insuficiente, o que la cantidad a retirar sea capturada incorrectamente. En el primer caso, el retiro se realiza con éxito; no así en los otros dos. En todos los casos se debe notificar el resultado de la operación. Definiendo el primer caso, donde el retiro se realiza con éxito:

$$
\begin{array}{|l}
\hline
\textit{RetiroOK} \\
\Delta CuentaBancaria \\
cantidadARetirar? : \mathbb{Z} \\
\hline
cantidadARetirar? > 0 \\
cantidadARetirar? \leqslant saldo \\
saldo' = saldo - cantidadARetirar? \\
\hline
\end{array}
$$

Ahora, se define el segundo caso, donde no hay suficiente saldo para realizar el retiro, y no se modifica el estado de la cuenta:

$$
\begin{array}{|l}
\hline
\textit{RetiroNoOK} \\
\Xi CuentaBancaria \\
cantidadARetirar? : \mathbb{Z} \\
\hline
cantidadARetirar? > 0 \\
cantidadARetirar? > saldo \\
\hline
\end{array}
$$

Ahora, se especifica el tercer caso, donde la cantidad es menor o igual a cero:

$$
\begin{array}{|l}
\hline
\textit{RetiroNoOK2} \\
\Xi CuentaBancaria \\
cantidadARetirar? : \mathbb{Z} \\
\hline
cantidadARetirar? \leqslant 0 \\
\hline
\end{array}
$$

Para los posibles resultados, se define un tipo libre de la siguiente manera:

$$RESULTADO = RetiroExitoso \mid SaldoInsuficiente \mid CantidadIncorrecta$$

Y se realiza la especificación de cada resultado:

$$
\begin{array}{|l}
\hline
\textit{ResultadoRetiroExitoso} \\
result! : RESULTADO \\
\hline
result! = RetiroExitoso \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ResultadoSaldoInsuficiente} \\
\hline
\textit{result}! : \textit{RESULTADO} \\
\hline
\textit{result}! = \textit{SaldoInsuficiente} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ResultadoCantidadIncorrecta} \\
\hline
\textit{result}! : \textit{RESULTADO} \\
\hline
\textit{result}! = \textit{CantidadIncorrecta} \\
\hline
\end{array}
$$

El escenario total de retiro está dado por:

$$
\begin{aligned}
\textit{Retiro} \quad \widehat{=} \quad & (\ \textit{RetiroOK} \wedge \textit{ResultadoRetiroExitoso}\ ) \vee \\
& (\ \textit{RetiroNoOK} \wedge \textit{ResultadoSaldoInsuficiente}\ ) \vee \\
& (\ \textit{RetiroNoOK2} \wedge \textit{ResultadoCantidadIncorrecta}\ )
\end{aligned}
$$

Obsérvese que para realizar la combinación de los esquemas, en lugar de utilizar el operador de conjunción ($\wedge$) explícitamente, también se podría obtener el mismo resultado si se hubieran incluido los nombres de los esquemas de los resultados dentro en los casos del retiro de la siguiente manera:

$$
\begin{array}{|l}
\hline
\textit{RetiroOK} \\
\hline
\Delta \textit{CuentaBancaria} \\
\textit{ResultadoRetiroExitoso} \\
\textit{cantidadARetirar}? : \mathbb{Z} \\
\hline
\textit{cantidadARetirar}? > 0 \\
\textit{cantidadARetirar}? \leqslant \textit{saldo} \\
\textit{saldo}' = \textit{saldo} - \textit{cantidadARetirar}? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RetiroNoOK} \\
\hline
\Xi \textit{CuentaBancaria} \\
\textit{ResultadoSaldoInsuficiente} \\
\textit{cantidadARetirar}? : \mathbb{Z} \\
\hline
\textit{cantidadARetirar}? > 0 \\
\textit{cantidadARetirar}? > \textit{saldo} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RetiroNoOK2} \\
\Xi \textit{CuentaBancaria} \\
\textit{ResultadoCantidadIncorrecta} \\
\textit{cantidadARetirar}? : \mathbb{Z} \\
\hline
\textit{cantidadARetirar}? \leqslant 0 \\
\hline
\end{array}
$$

Entonces, la especificación total estaría dada por:

$$\textit{Retiro} \;\; \widehat{=} \;\; \textit{RetiroOK} \vee \textit{RetiroNoOK} \vee \textit{RetiroNoOK2}$$

14.17.2. Observaciones para la combinación de esquemas

Para combinar esquemas es posible también utilizar otras conectivas lógicas como implicación ($\Rightarrow$), equivalencia ($\Leftrightarrow$) y negación ($\neg$). Sin embargo, cuando se realiza una combinación de esquemas en general, el especificador debe tomar en consideración que los esquemas deben estar normalizados, de lo contrario, la combinación ofrece resultados no esperados.

Un esquema está normalizado cuando en la sección de declaraciones se encuentran únicamente variables con sus tipos relacionados, y en la sección de predicados se encuentran todas las restricciones para esas variables. Este proceso permite ver claramente cuales variables tienen tipos iguales aunque pertenezcan a conjuntos diferentes. Cuando una declaración hace referencia directa a un tipo (y no a un conjunto) se le denomina firma.

Por ejemplo, el siguiente esquema no se encuentra normalizado, pues $\mathbb{N}$ no es un tipo sino un conjunto. Puede observarse que en la sección de declaraciones se está omitiendo el verdadero tipo: $\mathbb{Z}$ y se está agregando implícitamente la restricción de que $\textit{usuariosConectados} \in \mathbb{N}$.

$$
\begin{array}{|l}
\hline
\textit{SistemaDeRed} \\
\textit{usuariosConectados} : \mathbb{N} \\
\hline
\textit{usuariosConectados} < 100 \\
\hline
\end{array}
$$

El mismo esquema anterior pero ya normalizado es el siguiente:

SistemaDeRed

$usuariosConectados : \mathbb{Z}$

$usuariosConectados < 100$

$usuariosConectados \in \mathbb{N}$

Un esquema normalizado como éste puede ya ser combinado con otros. Los problemas que se presentan por la falta de normalización se deben a que los operadores lógicos actúan solamente a nivel de la sección de predicados, y si hay restricciones en la sección de declaraciones, el esquema resultante después de la combinación no las tomará en cuenta adecuadamente al relacionarlos con el operador lógico.

14.17.3. Renombrando variables

Los nombres de los elementos declarados en un esquema pueden ser cambiados por otros sin alterar el sentido de la especificación. Por ejemplo, suponer el siguiente esquema:

Deposito

$\Delta CuentaBancaria$

$cantidad? : N$

$saldo' = saldo + cantidad?$

La declaración:

$Deposito[\ cant? / cantidad?\ ]$

denota el esquema *Deposito* con la variable *cantidad?* renombrada por *cant?*, entonces el nuevo esquema queda así:

Deposito

$\Delta CuentaBancaria$

$cant? : N$

$saldo' = saldo + cant?$

Nótese la sintaxis para renombrar: entre los corchetes que siguen al nombre del esquema, se coloca el nombre nuevo de la variable, el caracter de barra y el nombre antiguo que se va a cambiar. Si se desean hacer varios cambios de nombre, se pueden indicar de esta misma manera, separados por comas dentro de los corchetes.

14.17.4. Ocultando variables

Es posible ocultar variables que han sido declaradas en un esquema utilizando el operador barra (****). Esto da como resultado que la variable ocultada deja de estar en la sección de declaraciones, y pasa a ser cuantificada como variable local en la sección de predicados del esquema. Por ejemplo, suponer el siguiente esquema:

Numeros

$a, b, c : \mathbb{Z}$

$a > b$

$b > c$

Entonces, *Numeros* \ *(a)* denota el esquema *Numeros* con la variable *a* oculta. El resultado es el siguiente esquema:

Numeros

$b, c : \mathbb{Z}$

$\exists\, a : \mathbb{Z} \bullet (a > b \wedge b > c)$

Nótese que la sintaxis requiere primero el nombre del esquema, luego el operador barra (también llamado de ocultamiento) y al final entre paréntesis el nombre de la variable que se va a ocultar. Si se desea ocultar más de una variable, se colocan sus nombres separados por comas.

14.17.5. Composición de esquemas

La composición de esquemas se realiza con el operador (⨾) y representa la unión de dos esquemas A y B, donde el estado de las variables comunes a ambos se combina de la siguiente manera: el estado *después* de las variables de A pasa a ser el estado *antes* de las variables de B. Las variables a combinar deben tener el mismo nombre en los esquemas A y B; la decoración no cuenta.

Por ejemplo, considérense las siguientes especificaciones para realizar exitosamente un depósito y un retiro en una cuenta bancaria:

$$
\begin{array}{|l}
\hline
\textit{CuentaBancaria} \\
\hline
saldo : \mathbb{Z} \\
\hline
saldo \geqslant 0 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{DepositoOK} \\
\hline
\Delta CuentaBancaria \\
cantidadADepositar? : \mathbb{Z} \\
\hline
saldo' = saldo + cantidadADepositar? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RetiroOK} \\
\hline
\Delta CuentaBancaria \\
cantidadARetirar? : \mathbb{Z} \\
\hline
cantidadARetirar? > 0 \\
cantidadARetirar? \leqslant saldo \\
saldo' = saldo - cantidadARetirar? \\
\hline
\end{array}
$$

Entonces, el estado de la cuenta al realizar de manera consecutiva y exitosa un depósito y un retiro, está definido por:

$$DepositoYRetiroOK \mathrel{\widehat{=}} DepositoOK \mathbin{⨾} RetiroOK$$

El esquema resultante es:

$$
\begin{array}{|l}
\hline
\textit{DepositoYRetiroOK} \\
\hline
\Delta CuentaBancaria \\
cantidadADepositar?, cantidadARetirar? : \mathbb{Z} \\
\hline
saldo + cantidadADepositar? \geq 0 \\
cantidadARetirar? > 0 \\
cantidadARetirar? \leqslant saldo + cantidadADepositar? \\
saldo' = saldo + cantidadADepositar? - cantidadARetirar? \\
\hline
\end{array}
$$

El resultado de componer dos esquemas de esta manera, se obtiene metodológicamente aplicando la definición para el operador de composición de esquemas, que se resume de la siguiente manera. Sean dos esquemas S, T, sea s la representación de las variables de estado *después* (′) en el esquema S que serán combinadas con sus respectivos estados *antes* en T, entonces S ⨾ T es el resultado obtenido al final de la siguiente lista de pasos consecutivos:

1) Obtener S[s+ / s’]
2) Obtener T[s+ / s]
3) Obtener S[s+ / s’] ∧ T[s+ / s’]
4) Obtener (S[s+ / s’] ∧ T[s+ / s’]) ∖ (s+)

14.17.6. Piping

Si se desea relacionar dos esquemas A y B, donde las salidas de A son justamente las entradas de B, se utiliza el operador de *piping* o de pipa (≫).

Por ejemplo, considerar los siguientes esquemas:

LeerDatos

$archivo? : LLAVE \longrightarrow DATOS$
$llave? : LLAVE$
$datos! : DATOS$

$datos! = archivo?\ llave?$
$llave? \in dom\ (archivo?)$

ColocarDatosEnRAM

$datos? : DATOS$
$estructuraMemoria! : DATOS$

$estructuraMemoria! = datos?$

Entonces:

$$LeerYColocarDatos \mathrel{\widehat{=}} LeerDatos \gg ColocarDatosEnRAM$$

denota la operación de leer datos de un archivo y luego colocarlos en una estructura en la memoria principal. El esquema resultante es el siguiente:

LeerYColocarDatos

$archivo? : LLAVE \longrightarrow DATOS$

$llave? : LLAVE$

$estructuraMemoria! : DATOS$

$llave? \in dom\ (archivo?)$

$estructuraMemoria! = archivo?\ llave?$

Obsérvese que las variables que se combinan deben tener el mismo nombre sin importar las decoraciones, y no aparecen en la versión final del esquema.

El resultado de combinar dos esquemas de esta manera, se obtiene metodológicamente aplicando la definición para el operador pipa, que se resume a continuación. Sean dos esquemas S, T, sea *y* la representación de las variables de estado de salida en S que corresponden a entradas en T, y sea p un nuevo nombre para *y*, entonces, $S \gg T$ es el resultado obtenido al final de la siguiente lista de pasos consecutivos:

1) Obtener S[p/ y!]
2) Obtener T[p/ y?]
3) Obtener S[p/ y!] $\wedge$ T[p/ y?]
4) Obtener (S[p/ y!] $\wedge$ T[p/ y?]) \ (p)

Bibliografía

ABRIAL, J. R. 1996. *The B-Book Assigning programs to meanings*. Cambridge Univeersity Press.

AGERWALA, Tilak. 1979. Putting Petri Nets to Work. *Computer*.

ALGORRY, Marcelo. 2005. Programación Orientada a Aspectos. *Revista Users.Code*.

ARAÚJO, João, Elisa BANIASSAD, Paul CLEMENTS et al. 2005. *Early Aspects: Current Landscape*.

ASTEASUAIN, Fernando and Bernardo CONTRERAS. 2002. *Programación Orientada a Aspectos: Análisis del Paradigma*.

ASTEASUAIN, Fernando, Bernardo CONTRERAS, Elsa ESTÉVEZ, and Pablo FILLOTTRANI. 2003. Programación Orientada a Aspectos: Metodología y Evaluación. *In*: *Proceedings of the IX CACIC 2003 - Congreso Argentino de Ciencias de la Computación*. La Plata.

ASTLEY, Mark and Gul AGHA. 1998. Modular Construction and Composition of Distributed Software Architectures. *In*: *International Symposium on Software Engineering for Parallel and Distributed Systems, 1998.*

BANACH, Richard. 2007. Guest Editorial: Formal Methods. *Computer Society of India Communications*, Mayo.

BANIASSAD, Elisa, Paul CLEMENTS, Joao ARAUJO et al. 2006. Discovering Early Aspects. *IEEE Software*. **23**(1), pp.61-70.

BARDEN, Rosalind, Susan STEPNEY, and David COOPER. 1994. *Z in practice*. Pearson Education.

B-CORE. 2008. *B-Core's website*. [online]. Available from World Wide Web: <http://www.b-core.com>

BECKER, Valeria and Claudia PONS. 2003. Definición formal de la semántica de UML-OCL a través de su traducción a Object-Z. *IX Congreso Argentino de Ciencias de la Computación CACIC*.

BELLINI, P., R. MATTOLINI, and P. NESI. 2000. Temporal Logics for Real-Time System Specification. *ACM Computing Surveys*. **32**(1).

BISHOP, Matt, Brian HAY, and Kara NANCE. 2011. Applying Formal Methods Informally. *In*: *Proceedings of the 44th Hawaii International Conference on System Sciences*.

BLAIR, G., L. BLAIR, A. RASHID et al. 2004. Engineering Aspect Oriented Systems. *In*: *Aspect-Oriented Software Development*, Adisson-Wesley Professional, pp.379-406.

BOWEN, Jonathan and Michael HINCHEY. 1995. Seven More Myths of Formal Methods. *IEEE Software*. **12**(4).

BRAUDE. 2008. *Ingeniería de software. Una perspectiva orientada a objetos*. México: AlfaOmega.

CHITCHYAN, Ruzanna, Awais RASHID, and Peter SAWYER. 2005. Comparing Requirements Engineering Approaches for Handling Crosscutting Concerns. *In*: *Conference on Advanced Information Systems Engineering*. Porto, Portugal.

CHITCHYAN, Ruzanna, Awais RASHID, Pete SAWYER et al. 2005. *Survey of Analysis and Design Approaches*.

CLELAND-HUANG, Jane. 2005. Software Requirements. *In*: *Software Engineering Volume 1: The Development Process*, pp.113-123.

CLEMENTE, Pedro, Juan HERNÁNDEZ, José HERRERO et al. 2004. Aspect-Orientation in the Software Lifecycle: Fact and Fiction. *In*: *Aspect-Oriented Software Development*, Addison-Wesley, pp.425-458.

COLYER, Adrian, Awais RASHID, and Gordon BLAIR. 2004. *On the Separation of Concerns in Program Families*. Lancaster, Inglaterra.

CONSTANTINIDES, Constantinos A., Atef BADER, and Tzilla ELRAD. 1999. A framework to address a two-dimensional composition of concerns. *In*: *First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems. OOPSLA'99*.

CWB, CCS and. 2008. *What is CCS and CWB?* [online]. Available from World Wide Web: <http://ei.cs.vt.edu/~cs5204/sp99/ccs.html>

DAVIES, Jim. 1993. *Specification and Proof in Real-Time CSP*. Cambridge University Press.

DE PAULA, Virginia and Thais BATISTA. 2007. Revisiting a Formal Framework for Modeling Aspects in the Design Phase. *In*: *Early Aspects at ICSE: Workshop in Aspect-Oriented Requirements Engineering and Architecture Design*.

DESOUZA, Kevin, Yukika AWAZU, and Amrit TIWANA. 2006. Four dynamics for bringing use back into software reuse. *Communications of the ACM*. **49**(1).

DIAZ, Rebeca. 2002. *Reutilización de requisitos funcionales de sistemas distribuidos utilizando técnicas de descripción formal*. Vigo, España.

DÍAZ PACE, J. Andrés and Marcelo R. CAMPO. 2001. Analyzing the Role of Aspects in Software Design. *Communications of the ACM*., pp.67-73.

DILLER, Antoni. 1999. *Z an introduction to Formal Methods*. Wiley.

EDLICH, Stefan, Reidar HÖRNING, Henrik HÖRNING, and Jim PATERSON. 2006. *The Definitive Guide to db4o*. Apress.

ELRAD, Tzilla, Omar ALDAWUD, and Atef BADER. 2004. Expressing Aspects Using UML Behavioral and Structural Diagrams. *In*: *Aspect-Oriented Software Development*, Addison-Wesley, pp.459-478.

ELRAD, Tzill and Robert E. FILMAN. 2001. Aspect-Oriented Programming. *Communications of the ACM*., pp.29-32.

ESTELLE. 2008. *Protocol Engineering Laboratory/University of Delaware*. [online]. Available from World Wide Web: <http://www.cis.udel.edu/~amer/PEL/estelle/index.html>

ESTUBLIER, Jacky and German VEGA. 2005. Reuse and Variability in Large software Applications. *ACM SIGSOFT - Software Engineering Notes*. **30**(5).

FAIRLEY, Richard. 1994. *Ingeniería de Software*. McGrawHill.

FERREIRA, Ricardo and Ricardo RAMINHOS. 2005. Metadata Driven Aspect Specification. *In*: *7th Interantional Workshop on Aspect-Oriented Modelling, MoDELS / UML*. Jamaica.

FIGUEROA, Pablo and Pablo ISAZA. 2006. Programación por Aspectos. Una Introducción.

FOKKINK, Wan. 2000. *Introduction to Process Algebra*. Springer.

GALÁN, José. 2003. Métodos formales orientados a objetos. *NOVATICA: Revista de la asociación de técnicos en informática*.

GARCÍA, Jorge. 2000. *Especificación, verificación y mantenimiento de requisitos funcionales con técnicas de descripción formal*. Vigo, España.

GIL, Alberto. 1999. *Diseño y verificación de sistemas distribuidos mediante la aplicación combinada de métodos formales*. Vigo, España.

GOGUEN, Joseph and Malcom GRANT. 1996. *Algebraic Semantics of Imperative Programs*. MIT Press.

GRADECKI, Joseph and Nicholas LESIECKI. 2003. *Mastering AspectJ - Aspect Oriented Programming in Java*. Wiley Publishing, Inc.

GREEN, Jeffrey. 1998. Correspondences between statecharts, event procedures, and concurrent regular expressions. *In*: *Proceedings of the 36th Annual Southeast Regional Conference*.

GROUP, University of Cambridge Computer Laboratory - Automated Reasoning. 2008. *Portal de HOL*. [online]. Available from World Wide Web: <http://www.cl.cam.ac.uk/research/hvg/HOL/>

GUTTAG, John, James HORNING, and Jeannette WING. 1985. The Larch Family of Specification Languages. *IEEE Software*.

HALL, Anthony. 1990. Seven Myths of Formal Methods. *IEEE Software*. **7**(5).

HAREL, David. 1986. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*., pp.231-274.

HARRISON, William, Harold OSSHER, and Peri TARR. 2002. *Asymmetrically vs. Symmetrically Organized Paradigms for software composition*. IBM.

HAYES, I. J., C. B. JONES, and J. E. NICHOLLS. 1994. Understanding the differences between VDM and Z. *Software Engineering Notes*., pp.75-81.

HOLMES, Reid. 2006. Unanticipated reuse of large-scale software features. *In*: *Proceedings of the 28th International Conference on Software Engineering*. Shangai, China.

INTERNATIONAL ORGANIZATION FOR STANDARIZATION. 1989. *Information Processing Systems - Open Systems Interconection - ESTELLE - A Formal Description technique based on an Extended State Transition Model*. Ginebra.

ISO/IEC. 1996. *EBNF standard document*.

ISO/IEC-13568. 2002. *Information technology — Z formal specification notation — Syntax, type system and semantics*.

ISODA, Sadahiro. 1995. Experiences of a software reuse project. *Journal of Systems and Software*.

JACKY, Jonathan. 2008. *The way of Z practical programming with formal methods*. New York: Cambridge.

JIANG, Li, Armin EBERLEIN, and Behrouz H. FAR. 2005. Combining Requirements Engineering Techniques –Theory and Case Study. *In*: *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*.

KICILLOF, Nicolás. 2008. *Programación Orientada a Aspectos*. [online]. Available from World Wide Web: <http://www.willydev.net/descargas/prev/AOP.PDF>

KNIGHT, John, Colleen DEJONG, Matthew GIBBLE, and Luis NAKANO. 1997. Why are Formal Methods not Used More Widely? *The Fourth NASA Langley Formal Methods Workshop*.

LADDAD, Ramnivas. 2003. *AspectJ in action Practical Aspect-Oriented Programming*. Manning Publications Co.

LADDAD, Ramnivas. 2003. Aspect-Oriented Programming will improve quality. *IEEE Software*. **20**(6), pp.90-91.

LAMSWEERDE, Axel Van. 2000. Formal Specification: a Roadmap. *In*: *Proceedings of the International Conference on Software Engineering*. Limerick, Ireland.

LAMSWEERDE, Axel Van. 2001. Building Formal Requirements Models for Reliable Software. *Lecture Notes in Computer Science*. **2043**.

LARCH. 2008. *Larch Home Page*. [online]. Available from World Wide Web: <http://nms.lcs.mit.edu/Larch>

LAWRENCE, Pfleeger. 2002. *Ingeniería de software - Teoría y práctica*. Prentice Hall.

LIGHTFOOT, David. 2001. *Formal Specification using Z*. New York: Palgrave Macmillan.

LINDEN, Frank van der, Klaus SCHMID, and Eelco ROMMES. 2007. *Software Product Lines in Action The Best industrial Practice in Product Line Engineering*. Springer.

LÓPEZ, Martín. 2006. *Soporte metodológico para la evolución de especificaciones formales. Aplicaciones en sistemas telemáticos*. Vigo, España.

LÓPEZ, Oscar, Miguel LAGUNA, and José MARQUÉS. 2001. Reutilización del software a partir de requisitos funcionales en el modelo de MECANO: Comparación de escenarios. *In*: *IV Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes de Software*. San José, Costa Rica.

LOUGHRAN, Neil, Awais RASHID, Weishan ZHANG, and Stan JAARZABEK. 2004. Supporting Product Line Evolution with Framed Aspects. *In*: *Workshop on Aspects, Components and Patterns for Infrastructure Software*.

MACHADO, Eduardo, Marcílio DA SILVA, and R. KLEBER. 2007. DigitalAssets Discoverer: Automatic Identification of Reusable Software Components. *In*: *OOPSLA '07*. Montreal, Canadá.

MANZANARES, Salvador. 2005. *Programación Orientada a Aspectos: Una experiencia práctica con AspectJ*.

MEIER, Silvio, Tobias REINHARD, Reinhard STOIBER, and Martin GLINZ. 2007. Modeling and Evolving Crosscutting Concerns in ADORA. *In*: *Early Aspects at ICSE: Workshop in Aspect-Oriented Requirements Engineering and Architecture Design*.

METHOD, Vienna Development. 2008. *Vienna Development Method*. [online]. Available from World Wide Web: <http://www.vienna.cc/evdm.htm>

MILI, Hafedh, Ali MILI, Sherif YACOUB, and Edward ADDY. 2002. *Reuse Based Software Engineering*. Wiley Inter-Science.

MILNER, Robin. 1980. *A calculus of communicating systems*. Springer-Verlag.

MORDECHAI, Ben-Ari. 2008. *Principles of the Spin model checker*. Springer.

MOREIRA, Ana, Awais RASHID, and Joao ARAÚJO. 2005. Multi-Dimensional Separation of Concerns in Requirements Engineering. *In*: *International Conference on Requirements Engineering*., pp.285-296.

MOUSAVI, Mohammad, Giovanni RUSELLO, Michel CHAUDRON et al. 2002. Aspects + GAMMA = AspectGAMMA A Formal Framework for Aspect-Oriented Specification. *In*: *Early Aspects: Aspect Oriented Requirements Engineering and Architecture Requirements Workshop*.

MURATA, Tadao. 1999. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*. **77**(4).

NAFEES QAMAR, N., Aamer NADEEM, and Uzair KHAN. 2006. An Automatable Framework for Formal Specification & Verification of Aspect Oriented Programs. *In*: *Workshop on Critical System Development Using Modeling Languages - ACM / IEEE Models Conference 2006*.

NAGEL, Christian, Bill EVJEN, Jay GLYNN et al. 2008. *Professional C# 2008*. Wiley.

NAVASA, A., K. PALMA, J. M. MURILLO, and Y. ETEROVIC. 2004. Dos modelos arquitectónicos para el DSOA. *In*: *Desarrollo de Software Orientado a Aspectos DSOA 2004*. Málaga.

NIPKOW, Tobias, Markus WENZEL, and Lawrence PAULSON. 2002. Isabelle/HOL A Proof Assistant for Higher-Order Logic.

NIU, Nan and Steve EASTERBROOK. 2008. Extracting and Modeling Product Line Functional Requirements. *In*: *16th IEEE International Requirements Engineering Conference*.

NUSEIBEH, Bashar. 2004. Croscutting Requirements. *In*: *Proceedings of the 3rd. International Conference on Aspect Oriented Software Development*. Lancaster, Inglaterra.

OBJ. 2008. *The OBJ Family*. [online]. Available from World Wide Web: <http://www.cs.ucsd.edu/users/goguen/sys/obj.html>

OSSHER, Harold and Peri TARR. 2000. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. *In*: *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.*

OSSHER, Harold and Peri TARR. 2001. Using Multidimensional Separation of Concerns to (re)Shape Evolving Software. *Communications of the ACM*., pp.43-50.

OWRE, Sam, John RUSHBY, and Natarajan SHANKAR. 1992. PVS: A Prototype Verification System. *In*: *Lecture Notes in Artificial Intelligence*. Springer-Verlag, pp.748-752.

PARKASH, Kamal and Rajesh BHATIA. 2007. Aspectual Reuse - Software Reuse and Aspect Oriented Requirement Engineering. *In*: *Proceedings of National Conference on Challenges & Opportunities in Information Technology*.

PARNAS, D. L. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*.

PIATTINI, Mario and Javier GARZÁS. 2007. *Fábricas de software: experiencias, tecnologías y organización*. México, D.F.: Alfaomega - Ra-Ma.

PINTO, M., L. FUENTES, and J. M. TROYA. 2003. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. *In*: *International Conference on GPCE*. Erfurt, Germany.

PRESSMAN, Roger. 2005. *Ingeniería del software. Un enfoque práctico*. McGrawHill.

PRIETO-DIAZ, Rubén. 1993. Status Report: Software Reusability. *IEEE Software*. **10**(3).

PRITCHARD, Jason. 1999. *COM and CORBA Side by Side Architectures, Strategies, and Implementations*. Addison Wesley Longman, Inc.

PROMELA. 2008. *Promela Language*. [online]. Available from World Wide Web: <http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html>

PRYWES, N. S., A. PNUELI, and S. SHASTRY. 1979. Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development. *ACM Transactions on Programming Languages and Systems*.

PVS. 2008. *portal de PVS*. [online]. Available from World Wide Web: <http://pvs.csl.sri.com/>

QUIROGA, Diego. 2004. Orientación a Objetos. *Revista Users.Code*, pp.58-62.

RAJAN, Hridesh. 2005. *Unifying Aspect- and Oject-Oriented Program Design*.

RASHID, Awais and Ruzanna CHITCHYAN. 2008. Aspect-Oriented Requirements Engineering: A roadmap. *In*: *Early Aspects 08*. Leipzig, Alemania, pp.35 - 41.

RASHID, Awais, Peter SAWYER, Ana MOREIRA, and Joao ARAUJO. 2002. Early Aspects: a Model for Aspect-Oriented Requirements Engineering. *In*: *IEEE Joint International Conference on Requirements Engineering*.

RAVICHANDRAN, T. and Marcus ROTHENBERGER. 2003. Software Reuse Strategies and Component Markets. *Communications of the ACM*. **46**(8).

REINA, Antonia. 2000. *Visión General de la Programación Orientada a Aspectos*. España.

ROBERTS, Nathan V., Eunjee SONG, and Paul C. GRABOW. 2009. Model Interfaces for Two-Way Obliviousness. *In*: *ACM Symposium on Applied Computing (SAC 2009)*. Honolulu, Hawaii, U.S.A.

ROCKSTRÖM, Anders and Roberto SARACCO. 1982. SDL - CCITT Specification and Description Language. *IEEE Transactions on Communications*.

RUMBAUGH, James, Ivar JACOBSON, and Grady BOOCH. 1999. *The Unified Modeling Language Reference Manual*. Addison Wesley.

RUSTY HAROLD, Elliotte. 1998. *JavaBeans*. IDG Books Worldwide.

SADAOUI, Samira and Pengzhou YIN. 2004. Generalization for Component Reuse. *In*: *Proceedings of the 42nd annual Southeast Regional Conference*. Alabama, USA.

SANTANA, Eduardo, Alexandre ALVARO, Vinicius CARDOSO et al. 2007. *Component Reuse in Software Engineering*. C.E.S.A.R. E-Books.

SCHAEFER, Ina and Reiner HÄHNLE. 2011. Formal Methods in Software Line Engineering. *IEEE Computer*.

SCHWARTZ, J. T., R. B. DEWAR, E. SCHONBERG, and E. DUBINSKY. 1986. *Programming with sets; an introduction to SETL*. Springer-Verlag.

SDL. 2008. *International Engineering Consortium*. [online]. Available from World Wide Web: <http://www.iec.org/online/tutorials/sdl>

SIOBHÁN, Clarke and Robert WALKER. 2004. Generic Aspect-Oriented Design with Theme/UML. *In*: *Aspect-Oriented Software Development*, Boston: Addison Wesley, pp.425-458.

SIY, Harvey, Prasanna ARYAL, Victor WINTER, and Mansour ZAND. 2007. Aspectual Support for Specifying Requirements in Software Product Lines. *In*: *Early Aspects at ICSE: Workshop in Aspect-Oriented Requirements Engineering and Architecture Design*.

SMITH, Graeme. 2001. State-Based Approaches: From Z To Object-Z. *In*: *Formal Methods for Distributed Processing*, Cambridge, pp.105-125.

SODHI, Jag and Prince SODHI. 1999. *Software reuse: Domain Analysis and Design Process*. McGrawHill.

SOMMERVILLE, Ian. 2005. *Ingeniería del Software*. Pearson - Addison Wesley.

SPIVEY, J. M. 1992. *The Z Notation: A Reference Manual*. Prentice Hall International (UK).

SUTTON, Stanley Jr. and Isabelle ROUVELLOU. 2004. Concern Modeling for Aspect-Oriented Software Development. *In*: *Aspect-Oriented Software Development*, Addison Wesley, pp.479-506.

TARR, Peri and Jr., Stanley M. SUTTON. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *In*: *The 21st. Conference on Software Engineering*. Los Angeles, California.

THAYER, Richard and Mark J. CHRISTENSEN. 2005. *Software Engineering Volume 1: The Development Process*. IEEE.

TROELSEN, Andrew. 2000. *Developer's workshop to COM and ATL*. Library of Congress Cataloging - in - Publication Data.

UBAYASHI, Naoyasu and Shin NAKAJIMA. 2007. Context-aware Feature Oriented Modeling with an Aspect Extension of VDM. *In*: *SAC '07*. Seul, Korea, pp.1269-1274.

UTTING, Mark. Julio, 2000. Data Structures for Z Testing Tools. *In*: *FM-TOOLS 2000*. Alemania.

VAN EIJK, P. H. J., C. A. VISSERS, and M. DÍAZ. 1989. *The formal description technique: LOTOS*. Elsevier Science Publishers.

VANOLI, Verónica, Sandra CASAS, and Claudia MARCOS. 2006. Un enfoque práctico para reusar aspectos en AspectJ. *In*: *XV Jornadas de Ingeniería del Software y Bases de Datos*. Barcelona, España.

VIENNEAU, R. 1996. A Review of Formal Methods. *Software Engineering IEEE Computer Society*.

WALKER, Robert and Gail MURPHY. 2000. Implicit Context: Easing Software Evolution and Reuse. *ACM SIGSOFT - Software Engineering Notes*. **25**(6).

WING, Jeannette. 1990. A Specifier's Introduction to Formal Methods. *IEEE Computer*. **23**(9).

WOODCOCK, Jim and Jim DAVIS. 1996. *Using Z Specification, Refinement, and Proof*. Prentice Hall.

YU, Huiqun, Dongmei LIU, Li YANG, and Xudong HE. 2005. Formal Aspect-Oriented Modeling and Analysis by AspectZ. *Proceedings of 17th International Conference on Software Engineering and Knowledge Engineering*.

ZAVE, Pamela. 1991. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*.