Departamento de Tecnologías y Sistemas de Información

Universidad de Castilla-La Mancha

Tesis Doctoral

Proceso de Evaluación basado en Pruebas para la Sustitución de Componentes Software

Andrés Pablo Flores

Área de Conocimiento: Lenguajes y Sistemas Informáticos

Ciudad Real España

Julio 2009

Departamento de Tecnologías y Sistemas de Información

Universidad de Castilla-La Mancha

Tesis Doctoral

Proceso de Evaluación basado en Pruebas para la Sustitución de Componentes Software

Andrés Pablo Flores

Área de Conocimiento: Lenguajes y Sistemas Informáticos

Director:

Dr. Macario Polo Usaola

Departamento de Tecnologías y Sistemas de Información,
Escuela Superior de Informática,
Universidad de Castilla-La Mancha.

Prefacio

Esta Tesis es presentada como parte de los requisitos para optar al grado académico de Doctor en Informática, de acuerdo al Programa: "Arquitectura y Gestión de la Información y del Conocimiento en Sistemas de Red", de la Escuela Superior de Informática, de la Universidad de Castilla-La Mancha y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras.

La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamento de Tecnologías y Sistemas de Información bajo la dirección del Dr. Macario Polo Usaola, Profesor Titular de Universidad, perteneciente al Departamento de Tecnologías y Sistemas de Información, de la Escuela Superior de Informática, de la Universidad de Castilla-La Mancha.

Andrés Pablo Flores

Departamento de Tecnologías y
Sistemas de Información
Escuela Superior de Informática
Universidad de Castilla-La Mancha

Resumen

El Desarrollo de Software basado en Componentes ha emergido para facilitar el ensamblaje de sistemas. Sin embargo la modificación de un sistema mediante la sustitución o actualización de componentes demanda un manejo cuidadoso debido a los riesgos de estabilidad en los sistemas ya desplegados. Los componentes de reemplazo deben ser apropiadamente evaluados para identificar si suministran el comportamiento esperado que ha sido afectado por la sustitución. Para resolver este problema, en esta tesis se propone un Proceso de Evaluación para la Sustitución de Componentes Software, en el que se complementa el análisis de compatibilidad que comúnmente se realiza, mediante la aplicación de criterios de cobertura de prueba de caja negra. El objetivo principal es observar el comportamiento de los componentes, que se logra analizando sus funciones internas de transformación de datos, lo cual satisface la métrica de pruebas *facilidad de observación*. El enfoque está conceptualmente basado en la técnica de pruebas denominada Back-to-Back. Cuando un componente debe ser reemplazado, se construye un conjunto de prueba (TS de *test suite*) con el propósito de representar los aspectos de comportamiento del componente, que denominamos TS de Comportamiento de Componente. Este TS es luego ejercitado contra las actualizaciones o componentes candidatos con el propósito de identificar la compatibilidad que se requiere. La automatización del proceso está inicialmente soportado en su totalidad mediante la herramienta *testooj*, en la cual el doctorando ha implementado cada una de las restricciones y condiciones de los pasos en el proceso, alcanzando de esta manera un mayor rigor y por lo tanto un enfoque más confiable.

Abstract

Software components have emerged to ease the assembly of software systems. However updating a system by substitution or upgrades of components demand a careful management since stability risks on deployed systems. Replacement components must be properly evaluated to identify whether they provide the expected behaviour affected by substitution. To address this problem, in this thesis the doctorand proposes a Process for Substitutability Assessment of Software Componentes, in which the regular compatibility analysis is complemented with the use of black-box testing coverage criteria. The main goal is to observe the components behaviour that is achieved by analysing their internal functions of data transformation, which fulfills the *observability* testing metric. The approach is conceptually based on the technique Back-to-Back testing. When a component should be replaced, a specific Test Suite TS is built in order to represent its behavioural facets, viz. a Component Behaviour TS. This TS is later exercised on candidate upgrades or replacement components with the purpose to identify the required compatibility. Automation of the process is supported through the tool *testooj*, which constraints conditions and steps of the whole process in order to provide a rigorous and reliable approach.

Agradecimientos

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Motivación

1.1. Introducción

El Desarrollo del Software basado en Componentes (DSBC) ha emergido como un enfoque de ingeniería concebido para acelerar la construcción de sistemas mediante un proceso que desplaza el esfuerzo de desarrollo tradicional por la selección e integración de componentes software [Cechich y otros, 2003].

El enfoque tradicional de construcción de sistemas involucra generalmente el desarrollo de cada una de sus piezas desde cero y ad-hoc para el proyecto en curso, aunque algunas de tales piezas a menudo provienen de proyectos previos y son reutilizadas mediante los ajustes necesarios. Tales ajustes posiblemente alteran algunos aspectos de interconexión o incluso parte de las funciones internas, al contar con el código fuente de tales piezas.

Esta tendencia hacia la reutilización ha conducido a conformar el proceso de DSBC que concentra su esfuerzo en la composición de piezas ya desarrolladas, definidas como componentes software, los cuales son generalmente autocontenidos en funcionalidad, aunque algunos pueden requerir de una vinculación a otros componentes para proveer una funcionalidad completa – quizá a un nivel más alto de abstracción [Gamma y otros, 1995; Szyperski, 2002]. Esa conexión se realiza asumiendo los componentes como cajas negras, dado que en general se utilizan "tal y como son encontrados", sin considerar alguna modificación interna para suponer la posibilidad de su uso, lo cual implica que invariablemente se requiera alguna clase de ajuste externo para lograr

esa composición [D'Souza y Wilis, 1998]. De esta manera, la fase de integración ya no implica el esfuerzo final del desarrollo, sino que se adelanta en el ciclo de vida tradicional, para así manipular los componentes involucrados con el fin de ensamblarlos para formar estructuras de mayor envergadura que permitan constituir finalmente el sistema software [Brown y Wallnau, 1996; Tran y otros, 1997; Cechich y otros, 2003].

El proceso de desarrollo tradicional se modifica en DSBC para incorporar tareas de selección y evaluación de componentes, que se basan principalmente en los requerimientos específicos de integración de un sistema, lo cual además influye considerablemente sobre la decisión de qué componentes pueden ser adquiridos de fuentes externas y cuáles de ellos deben ser desarrollados desde cero.

Para comprender adecuadamente el rol central de la integración en DSBC, se muestra en la Figura 1.1 un modelo de referencia sugerido por Brown y Wallnau [1996] que describe el tratamiento que se efectúa sobre los componentes, y permite intuir los desafíos a enfrentar para lograr que el proceso de DSBC sea eficiente y efectivo, tanto a nivel de la aceleración del proceso de desarrollo como de la reducción de esfuerzo y costes. A continuación se explican brevemente los estados que asumen los componentes a través del proceso de desarrollo:

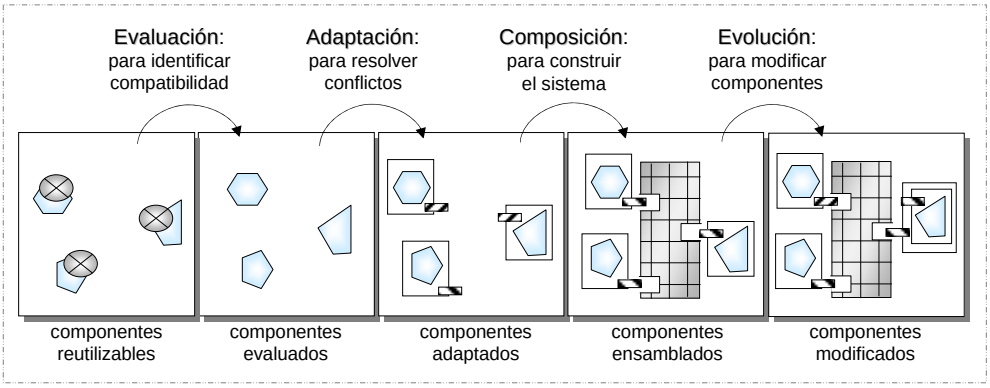


Figura 1.1: Modelo de Referencia para la Integración de Componentes

a. *Componentes reutilizables*: poseen diversos aspectos ocultos, principalmente en relación con la implementación de sus interfaces y el comportamiento requerido.

b. *Componentes evaluados*: han pasado ya por el proceso de identificación de los aspectos que pueden generar conflictos.

c. *Componentes adaptados*: se han solucionado las fuentes de conflicto, por medio de algún tipo de encapsulamiento o "*wrapper*".

d. *Componentes ensamblados*: ya han sido integrados al sistema, mediante alguna infraestructura arquitectónica.

e. *Componentes modificados*: han sido reemplazados por nuevas versiones o por componentes diferentes que presentan comportamiento e interfaces similares. Quizá esto implique la modificación de los wrappers.

Tras haber introducido brevemente algunos aspectos de la integración en el proceso de DSBC, a continuación se detallan algunos conceptos fundamentales que permiten conocer los desafíos a los que se enfrenta un ingeniero de sistemas. Esto proveerá un escenario de las áreas que aún requieren del desarrollo de estrategias adecuadas y permitirá definir los objetivos de esta tesis que se presentan en la Sección 1.2.

1.1.1. Componentes Software

Dado que existen diversas tecnologías que tratan el concepto de componentes software de manera diferente, es necesario adoptar una definición unificada, la cual surge como fusión de las definiciones dadas por Szyperski [2002] y Gross [2005]:

> "Un componente es una unidad reutilizable de composición, con interfaces y cualidades explícitamente especificadas, que denota una abstracción simple y puede ser sujeto de composición por terceras partes, sin necesidad de modificación".

De la definición previa, surgen algunos conceptos complementarios que requieren una descripción extendida:

Interfaces Un componente encapsula un conocimiento y comportamiento específicos que sólo es accesible a través de sus interfaces. Cada interfaz define una colección de operaciones que especifica uno de los servicios del componente. Esto indica que existe una clara separación entre las interfaces y la implementación (encapsulada) del comportamiento de un componente. Así las *interfaces* actúan como puntos de acceso para el componente, a través de las cuales los servicios declarados pueden ser requeridos

por los clientes del componente [Kruchten, 1998; Stuckenholz, 2005; Kung-Kiu y Zheng, 2007].

Las interacciones con las interfaces de los componentes pueden ser identificadas como consecuencias de la ocurrencia de eventos. Un *evento* es por lo tanto un incidente en el cual el efecto resultante es la invocación de una interfaz. Particularmente, las interacciones entre dos componentes ocurren por eventos provocados por un componente y atendidos por otro, que se denominan eventos externos. El incidente puede ser ocasionado por una interfaz diferente, a través de una *excepción* o simplemente a través de la acción de un usuario tal como presionar un botón. En general un evento puede ser definido como la invocación de una interfaz a través de otra interfaz [Wu y otros, 2003; Jaffar-UrRehman y otros, 2007].

Procedencia Dado que los componentes están completamente desarrollados y se encuentran disponibles para su reutilización, se los conoce comúnmente como *off-the-shelf* (OTS). Los componentes OTS poseen una variada procedencia: aquellos desarrollados en algún proyecto previo, denominados *in-house*, entre los cuales se pueden mencionar los sistemas heredados (*legacy*), y en conjunto con los componentes OTS de código abierto (*open source*) presentan la particularidad de la accesibilidad al código fuente; mientras que otros componentes OTS que se adquieren de terceras partes (usualmente de proveedores comerciales), denominados COTS (*commercial-off-the-shelf*), no cuentan en general con tal disponibilidad de código fuente.

Esta variedad en la procedencia de los componentes resulta la mayor ventaja del proceso de DSBC, aunque también involucra el desafío más serio, dado que los componentes poseen "marcas" diferentes (atributos desconocidos y cualidades variadas), las cuales se ponen de manifiesto cuando se decide su utilización en un contexto que no fue inicialmente anticipado por el productor del componente. Por lo tanto, la selección y evaluación de componentes son actividades clave que se llevan a cabo en etapas tempranas del ciclo de vida de DSBC [Brown y Wallnau, 1996].

Modelo de Componentes Existe una amplia diversidad de tecnologías involucradas en la construcción de sistemas basados en componentes, cada una con requisitos y restricciones particulares que establecen el aspecto concreto que adopta un componente

y los mecanismos para su integración en un sistema software. El *Modelo de Componentes* define la estructura y morfología de los componentes, como así también las reglas de creación, composición y comunicación de los mismos [Kung-Kiu y Zheng, 2007].

El éxito de insertar o enchufar (*plug-in*) un componente en una composición, depende de la declaración precisa que provee el componente acerca de sus expectativas sobre los componentes a los cuales se lo conecta. Una composición de componentes se define como la combinación de uno o más componentes software que conforman un nuevo comportamiento a un nivel diferente de abstracción. Las características del nuevo comportamiento están determinadas no sólo por los componentes involucrados, sino también por la forma en que éstos están combinados.

Cada tecnología define mecanismos específicos de integración de componentes software mediante una *infraestructura de componentes* subyacente que permite la ejecución y comunicación de un conjunto interactivo de componentes software, asegurando que un sistema software o subsistema construido usando esos componentes e interfaces pueda satisfacer las especificaciones de rendimiento definidas [Heineman y Council, 2001; D'Souza y Wilis, 1998].

1.1.2. Ciclo de Vida para DSBC

Si bien el ciclo de vida tradicional del software no se puede aplicar en forma directa en el contexto de DSBC, se podría realizar una simplificación inicial de las etapas involucradas de acuerdo al modelo en cascada para comprender los cambios significativos que se generan debido al extenso uso de los componentes OTS existentes. La Figura 1.2 muestra una vista simplificada del ciclo de vida para DSBC [Crnkovic y otros, 2006], donde se observa que en la etapa de implementación se aplican las actividades presentadas en el modelo de referencia de la Figura 1.1 (pág. 2).

Una vez que se conoce una gran mayoría de los requerimientos funcionales de un sistema, una organización puede armar un inventario de componentes conocidos (locales y externos), para analizar si se pueden cubrir las partes de esa funcionalidad. Sin embargo al intentar efectuar la integración de la colección de componentes que deben satisfacer tales requisitos, surge una gran cantidad de desafíos diversos. Uno de tales desafíos implica decisiones de "adquisición" versus "desarrollo" con respecto a muchas de las partes del sistema. La clave radica en entender de qué manera balancear o compensar esos

desafíos, registrar el conocimiento surgido de la toma de decisiones, y evaluar cómo esas compensaciones afectarán el producto resultante [Brown y Wallnau, 1996].

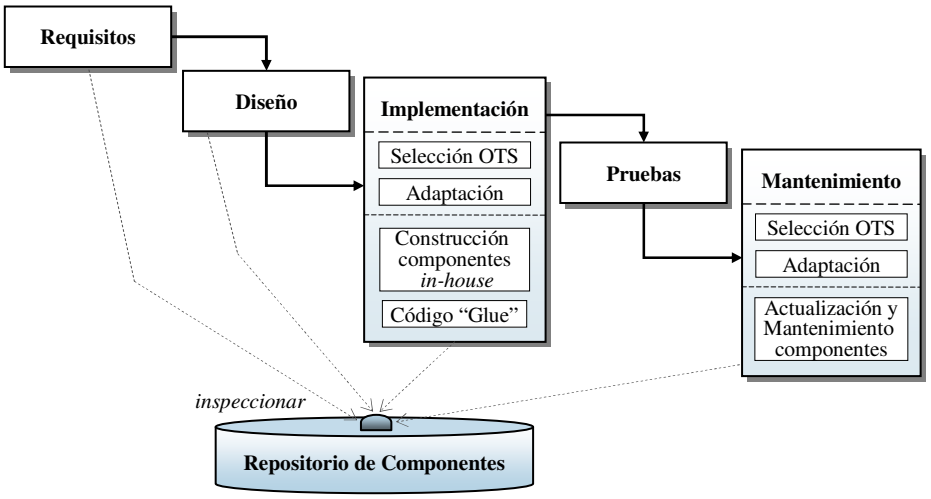


Figura 1.2: Vista simplificada del Ciclo de Vida para DSBC

En la práctica, esos desafíos pueden generar una desconfianza en algunas organizaciones, al suponer que no tendrán la capacidad de manejar un conjunto demasiado diverso de componentes. Por ello, si bien la selección de componentes debería incluir una revisión exhaustiva del mercado por todo posible componente que pudiera ser de interés, muchas organizaciones restringen esta actividad a un número reducido de componentes, y generalmente usan aquellos componentes con los cuales ya se encuentran familiarizados [Heineman y Council, 2001; Warboys y otros, 2005].

Como resultado de la selección inicial de componentes, la regla típica "80/20" algunas veces se aplica, es decir que el 80 % de la funcionalidad del sistema se puede solucionar de forma relativamente fácil con los componentes seleccionados, mientras que para el 20 % restante existe una mayor dificultad. En este punto los ingenieros de sistemas a menudo analizan ciertos cambios en los requisitos del sistema, que puedan facilitar un uso más efectivo de los componentes seleccionados. Tales cambios, una vez identificados, deben ser discutidos (o "negociados") con los usuarios del sistema para comprender la prioridad de esos requisitos y decidir cuales de ellos pueden ser ajustados.

Todas las decisiones que se toman y registran durante las etapas del desarrollo suministran la base para que en la etapa de mantenimiento sea factible aplicar mejoras al sistema a medida que los requisitos o las condiciones operativas evolucionan con el

tiempo. Como se observa en la Figura 1.2, el mantenimiento requiere desarrollar algunas de las actividades de la etapa de implementación, aunque de hecho para cualquier mejora aplicable estarían involucradas la mayoría de las actividades del ciclo de vida (desde requisitos hasta pruebas) [Crnkovic y otros, 2006].

1.1.3. Sustitución de Componentes

Una vez que un sistema basado en componentes ha sido exitosamente instalado y se encuentra en funcionamiento, ingresa ineludiblemente en su etapa de mantenimiento, dado que al igual que cualquier otro producto software el sistema estará sujeto a constantes modificaciones dada la naturaleza evolutiva del software. No solamente pueden ocurrir los cambios clásicos relacionados con los requisitos funcionales o no-funcionales del sistema, sino que en particular en un sistema basado en componentes algunas de sus piezas constituyentes (es decir los componentes) pueden también requerir una evolución que genera nuevas versiones o nuevas entregas (*releases*). Esta última situación que requiere el reemplazo de los componentes en el sistema genera una inquietud considerable acerca de la posibilidad de afectar la estabilidad actual del sistema [Cechich y otros, 2003; Jaffar-UrRehman y otros, 2007].

La facilidad de sustitución comprende un aspecto delicado para la actualización de cualquier producto software dada la dificultad de controlar el impacto de los cambios. En particular para DSBC algunos de los casos de modificaciones a algún componente pueden aún generar versiones compatibles debido a que sólo se corrigen algunos errores (*bugs*) que pasaron desapercibidos previo a disponer la entrega del componente, con lo cual todavía es posible un cierto control sobre los cambios producidos. Sin embargo cuando las versiones requieren que sean presentadas como nuevas entregas es posible que la magnitud de los cambios se haya generalizado y que los mismos se hayan propagado a través de la mayoría de las funciones y estructuras codificadas, afectando caminos estables de ejecución, y por lo tanto produciendo una diferencia masiva con respecto al componente original.

Este hecho es aún mas complicado cuando los componentes se adquieren de diferentes vendedores o proveedores de software, donde los ingenieros de software no pueden controlar el desarrollo y son incapaces de asegurar que el mismo entorno (como el compilador, y las opciones de compilación, entre otros) fueron usados en aquellos componentes que se los supone similares. Esta situación también se aplica a las propias entregas sucesivas de un mismo componente, dado que por ejemplo, las entregas que

incorporan nuevas tecnologías o nuevas soluciones arquitectónicas tienen una tendencia muy pronunciada a introducir un gran número de diferencias estructurales. Además es frecuente que los vendedores dispongan nuevas entregas de componentes populares para mantener la competitividad de sus productos en el mercado, como por ejemplo: Crystal Report[1] que permite la creación de reportes de impresión, y está sujeto a modificaciones bajo un esquema mensual [Mariani y otros, 2007; Stuckenholz, 2005].

En consecuencia, los ingenieros de software se ven forzados a aceptar las nuevas entregas de un componente o considerar el uso de otros componentes equivalentes de reemplazo, para mantener los sistemas actualizados y competitivos. Aunque, con cada reemplazo deben soportar la sobrecarga típica de tareas relacionadas con las pruebas de unidad y de integración, ocasionando que la reducción de esfuerzo, que era una de las promesas del paradigma de DSBC, no se cumpla convenientemente.

Por tal razón, la preocupación principal a ser cubierta en esta tesis es proveer un enfoque que pueda ayudar a los ingenieros de software a mantener la estabilidad de los sistemas basados en componentes, pero además considerando los aspectos prácticos que permitan simplificar algunas de sus tareas cotidianas. La intención es que los ingenieros de software cuenten con un Proceso de Evaluación como un soporte para la Facilidad de Sustitución, con el que sea posible identificar si nuevas entregas o nuevos componentes adquiridos pueden, de forma segura, reemplazar componentes actuales en sistemas basados en componentes ya desarrollados, desplegados y en uso.

El enfoque que se propone en esta tesis define el proceso de evaluación de componentes software particularmente considerando los aspectos funcionales del componente, tanto a nivel sintáctico (mediante el análisis de las interfaces de los componentes), como a nivel semántico (mediante el análisis del comportamiento de los componentes).

En particular para observar el comportamiento de los componentes se analizan sus funciones internas de transformación de datos, lo cual satisface la métrica de pruebas *facilidad de observación*, que se enfoca en el comportamiento operacional de un componente distinguiendo su salida como función de su entrada [Freedman, 1991; Jaffar-UrRehman y otros, 2007]. El análisis de los datos de entrada y salida, y de cómo los datos se transforman de unos en otros, proporciona un medio significativo para comparar el comportamiento de los componentes, como se discute en [Alexander y Blackburn, 1999; Cechich y Piattini, 2007].

[1] http://www.businessobjects.com

El enfoque está conceptualmente basado en la técnica de Pruebas Back-to-Back, la cual juzga la corrección de una nueva implementación en función de la generación de un conjunto de casos de prueba (TS, de *test suite*) para una implementación anterior (de referencia), y ejecutar ese TS contra ambas implementaciones para comparar los resultados [Vouk y otros, 1987; Shimeall y Leveson, 1991; Edwards, 2001; Veenendaal, 2006].

Así en nuestro enfoque, en cuanto surge la necesidad de que un componente deba ser reemplazado, se construye un TS específico cuyo objetivo es representar el comportamiento del componente, denominado TS de Comportamiento de Componente. Para el diseño de este TS se ha realizado una selección de criterios de cobertura de pruebas para componentes, identificando las estrategias efectivas para su implementación dentro del enfoque.

Luego se necesita realizar una búsqueda de componentes candidatos, los cuales serán ejercitados contra el TS de Comportamiento de Componente, para identificar si existe un cierto grado conveniente de compatibilidad con respecto al componente que debe ser reemplazado. Cuando más de un componente candidato ha sido considerado para su evaluación, el proceso además permite efectuar *selección de componentes* en función de la información que se genera a través de los diversos análisis a los que se someten los componentes candidatos.

Dado que el rigor de un proceso permite alcanzar un cierto grado de fiabilidad, se han automatizado los pasos que definen el proceso de evaluación mediante la herramienta *testooj* [Polo y otros, 2007, 2008], lo cual permite además generar condiciones de experimentación en cuanto a la eficacia y eficiencia del proceso. Si bien la herramienta *testooj* está particulamente enfocada en la prueba de componentes Java, su desarrollo y utilización generan un conocimiento valioso para explorar la implementación del proceso de evaluación con respecto a otros frameworks de componentes.

En función de lo expuesto anteriormente se definen a continuación los objetivos concretos de esta tesis, y se describen los proyectos de investigación que le sirvieron de marco. Luego de ello se presenta en la Sección 1.3 un resumen del método empleado para llevar adelante la investigación desarrollada y su aplicación al contexto específico de esta tesis. Finalmente se presenta en la Sección 1.4 la organización del contenido del resto de la tesis.

1.2. Hipótesis y Objetivos

La *Hipótesis* de trabajo para esta tesis es:

> ***Es factible efectuar el reemplazo de un componente software en un sistema basado en componentes, manteniendo su estabilidad actual, y asegurando dicha estabilidad con técnicas de prueba***

Basándonos en esta hipótesis se ha formulado el *Objetivo* principal de esta tesis de la siguiente manera:

> DEFINIR UN PROCESO DE EVALUACIÓN PARA LA SUSTITUCIÓN DE COMPONENTES SOFTWARE BASADO EN TÉCNICAS DE PRUEBA

Para la consecución de este objetivo planteamos los siguientes objetivos parciales:

OP.1. Analizar el estado del arte en cuanto a métodos y técnicas para la sustitución y evaluación de componentes software.

OP.2. Analizar el estado del arte en cuanto a técnicas de prueba de componentes software.

OP.3. Definir estrategias de comparación de interfaces de componentes, e implementar prototipos para llevar a cabo la experimentación.

OP.4. Definir estrategias de comparación de comportamiento de los componentes, e implementar prototipos para llevar a cabo la experimentación.

OP.5. Definir un proceso de evaluación para la sustitución de componentes software.

OP.6. Validar el enfoque mediante casos de estudio apropiados.

OP.7. Desarrollar una herramienta que asista en la aplicación del proceso.

1.2.1. Marco de la Tesis

Los proyectos de I+D con financiación en alguna convocatoria oficial de ámbito regional, nacional o iberoamericano, que han servido de marco para el desarrollo del trabajo de esta tesis, son los siguientes:

1.2.1.1. Proyectos Iberoamericanos

COMPETISOFT: *Mejora de Procesos para Fomentar la Competitividad de la Pequeña y Mediana Industria del Software de Íbero América*, Nº 506PI287, subprograma de CYTED (Ciencia y Tecnología para el Desarrollo). El proyecto estaba conformado por 16 grupos de investigación de Universidades Iberoamericanas, 2 entidades públicas Argentinas, y 5 empresas Iberoamericanas. Su inicio fue en el año 2006 y ha finalizado en diciembre de 2008, siendo liderado por el Dr. Mario Piattini Veltuis de la Universidad de Castilla-La Mancha y la Dra. Hanna Oktaba de la Universidad Nacional Autónoma de México. Su objetivo ha sido el incremento del nivel de competitividad de las micro, pequeñas y medianas empresas iberoamericanas productoras de software, mediante la creación y difusión de un marco metodológico que, ajustado a las necesidades específicas de cada empresa, fuera la base de un mecanismo de evaluación y certificación de la industria del software.

http://alarcos.inf-cr.uclm.es/Competisoft/

RITOS2: *Segunda Red Ibero-Americana de Tecnologías de Software para la década del 2000*, Proyecto CyTED (Ciencia y Tecnología para el Desarrollo) VII-J-RITOS2, compuesto por más de 30 grupos de investigación de España, Portugal y Latinoamérica. El proyecto, financiado por el Ministerio de Educación y Ciencia de España y dirigido por la Dra. Nieves Rodríguez Brisaboa de la Universidade da Coruña, finalizó en diciembre de 2004. Los temas de investigación principales de este proyecto estaban relacionados con ingeniería del software, sistemas distribuidos, procesamiento de lenguaje natural y bases de datos. Los objetivos principales de esta red se centraron en facilitar el desarrollo de tecnologías de Ingeniería del Software y potenciar su transferencia a los sectores industriales, de servicios y administraciones públicas; así como potenciar la formación de investigadores y de profesionales y ser foro de intercambio de experiencias, conocimientos, informaciones, bibliografía, documentación, etc.

http://emilia.dc.fi.udc.es/Ritos2/index.html

1.2.1.2. Proyectos Nacionales

MÁS: *Mantenimiento Ágil del Software*, Nº: TIC2003-02737-C02-02. Ha sido dirigido por el Dr. Macario Polo Usaola de la Universidad de Castilla-La Mancha, y se ha centrado

en el desarrollo de técnicas para la mejora del mantenimiento del software, incluyendo la evolución de los sistemas basados en componentes y aquellos basados en servicios web. Es un subproyecto de AGILWEB: *Desarrollo y Mantenimiento Ágil de Aplicaciones Basadas en Servicios Web*, liderado por el Dr. Miguel Toro de la Universidad de Sevilla, con el fundamento en la tecnología de los servicios web y en la carencia de metodologías, técnicas y herramientas existentes que garanticen la calidad en el desarrollo de aplicaciones basadas en los servicios web, o aquellas que poseen una tecnología similar como los sistemas basados en componentes.

http://alarcos.inf-cr.uclm.es/interfaces/spa/proyectos/proyectos.aspx

IEUCSoft: *Identificación, Evaluación y Uso de Composiciones Software*, Nº 04/E072. Proyecto que ha iniciado en 2008 y finalizará en 2010, siendo liderado por la Dra. Alejandra Cechich de la Universidad Nacional del Comahue en Argentina. Dentro de los alcances del proyecto se incluye la identificación y evaluación de componentes tanto en sistemas basados en componentes, como en aplicaciones orientadas a servicios.

http://giisco.uncoma.edu.ar

MPDSBC: *Mejora del Proceso de Desarrollo de Software Basado en Componentes*, Nº 04/E059. Proyecto que inició en 2005 y finalizó en 2007, siendo dirigido por la Dra. Alejandra Cechich de la Universidad Nacional del Comahue en Argentina. Dentro de los alcances del proyecto se incluye la evaluación de calidad en sistemas basados en componentes, y el proceso de mantenimiento de sistemas basados en componentes.

http://giisco.uncoma.edu.ar

MCDOO: *Modelado de Componentes Distribuidos Orientados a Objetos*, Nº 04/E048. Proyecto que inició en 2002 y finalizó en 2004, siendo liderado por el Dr. Ernesto Pimentel Sanchez de la Universidad de Málaga y la Dra. Alejandra Cechich de la Universidad Nacional del Comahue en Argentina. Su objetivo ha sido la investigación sobre el modelado de componentes cooperativos, desarrollo de software basado en componentes y evaluación de calidad en sistemas basados en componentes.

http://giisco.uncoma.edu.ar

1.2.1.3. Proyectos Regionales

PRALIN: *PRuebas pAra LINeas de Producto*, Nº PAC-08-0121-1374. Proyecto financiado por la Junta de Comunidades de Castilla-La Mancha, Consejería de Educación y Ciencia, siendo liderado por el Dr. Macario Polo Usaola, de la Universidad de Castilla La-Mancha. El proyecto ha iniciado en 2008 y tendrá duración hasta 2010. Los objetivos de este proyecto se centran en avanzar un paso más en la propuesta de técnicas para automatizar el desarrollo de software en el ámbito de las líneas de producto, dedicando el esfuerzo de investigación a la fase de pruebas, lo que incluye aspectos de generación de casos de prueba a partir de diferentes artefactos, propuesta de medidas de cobertura en función de los artefactos, mantenimiento de la trazabilidad e implementación de herramientas, teniendo presente las necesarias características de reutilización de las líneas de producto.

http://alarcos.inf-cr.uclm.es/interfaces/spa/proyectos/proyectos.aspx

1.3. Método de Investigación

En esta sección se presenta un resumen del método de investigación que ha sido aplicado en el desarrollo del trabajo de esta tesis, el cual se denomina *Método Investigación-Acción*. Luego se explica cómo se ha aplicado este método para alcanzar los objetivos que se definieron para conformar el trabajo de esta tesis.

1.3.1. El Método de Investigación-Acción

Actualmente existen diferentes métodos de investigación cualitativa, entre los que destaca el método Investigación-Acción. El término "Investigación-Acción" proviene del autor Kart Lewin [1947] que lo utilizaba para describir una forma de investigación que pudiera enlazar el enfoque experimental de las ciencias sociales con programas de acción social que respondieran a los principales problemas sociales de aquella época. Mediante la investigación-acción, Lewin argumentaba que se podían lograr en forma simultánea avances teóricos y cambios sociales. A pesar de que la primera propuesta de Investigación-Acción fue introducida en 1985 [Wood-Harper, 1985], en los últimos años ha alcanzado una gran atención y aceptación por parte de la comunidad investigadora

en Sistemas de Información [Avison y otros, 1999; Seaman, 1999].

Tomando las definiciones dadas por McTaggart [1991] y Wadsworth [1998] se puede describir la Investigación-Acción como: "la investigación que organiza condiciones bajo las cuales involucrar grupos de personas que son parte relevante de una determinada situación actual, con el objetivo de examinarla (por experimentarse como problemática), cambiarla y mejorarla, pero además aprender de sus propias experiencias y hacer esa experiencia accesible a otros".

De lo cual se puede argumentar que la Investigación-Acción tiene una doble finalidad: generar un beneficio al *cliente* de la investigación mediante la mejora de una situación problemática, y al mismo tiempo generar conocimiento que sea útil tanto para ellos como para otros.

En realidad Investigación-Acción no se refiere a un método de investigación concreto, sino a una clase de métodos que poseen una serie de características en común, las cuales además destacan claramente su utilidad con respecto a la Ingeniería del Software [Baskerville, 1999]:

- Orientación a la acción y al cambio en sistemas sociales: esto mejora el conocimiento de una situación social inmediata, enfatizando la naturaleza compleja y multi-variable de lo social en el dominio de la Ingeniería del Software.
- Focalización en un problema: permite asistir en la resolución de problemas prácticos y expande el conocimiento científico
- Posee un modelo de proceso orgánico que engloba etapas sistemáticas y algunas veces iterativas
- Se realiza cooperativamente y, como resultado de la colaboración, mejora la habilidad de los participantes

La Investigación-Acción combina teoría y práctica, investigadores y practicantes; mediante el cambio y la reflexión, dado que los resultados de esta experiencia deben ser beneficiosos tanto para el investigador como para los practicantes, y tal combinación se consigue gracias a la intervención de un investigador en la realidad del grupo en cuestión [Avison y otros, 1999]. Una premisa fundamental en esta forma de investigar es que los procesos sociales complejos (y el uso de tecnologías de la información en

organizaciones de este tipo) pueden ser mejor abordados y estudiados si se introducen cambios en dichos procesos y se observan los efectos de dichos cambios [Baskerville, 1999].

En el campo de los Sistemas de Información, el cliente de una investigación es normalmente una organización a la cual el investigador suministra servicios (tales como consultoría, ayuda para cambiar o desarrollar software), a cambio de tener acceso a datos de interés para la investigación y quizá para recibir financiación [Kock y Lau, 2001]. No obstante, el investigador que utiliza Investigación-Acción en Sistemas de Información (IA-SI) se enfrenta al desafío de cubrir las demandas tanto del cliente de la investigación como de la comunidad científica de Sistemas de Información. Las demandas de ambos suelen ser muy diferentes y a veces opuestas entre sí, sin embargo al satisfacer tanto las necesidades de los practicantes como las del conocimiento científico, se añaden nuevos elementos a la investigación que la enriquece de manera considerable.

En este método pueden identificarse cuatro tipos diferentes de roles, aún cuando algunos de ellos pueden ser cubiertos por la misma persona en simultáneo [Wadsworth, 1998]:

- Investigador: el individuo o el grupo que lleva a cabo, de forma activa, el proceso investigador.
- Objeto investigado: el problema a resolver.
- Grupo crítico de referencia: aquél para quien se investiga, en el sentido de que es quien tiene un problema que necesita ser resuelto. Participa en el proceso de investigación, pero menos activamente que el investigador. De este grupo participan personas que saben que son parte de la investigación y otras que no (por ejemplo pacientes sometidos a un tratamiento con placebo).
- Beneficiario: aquél para quien se investiga, en el sentido que es quien puede beneficiarse del resultado de la investigación (*stakeholder*) aunque no participe directamente en el proceso. Puede ser el receptor de documentos, informes, etc. En este grupo, por ejemplo, caben tanto las empresas que se benefician de un nuevo método para resolver problemas en tecnologías de la información, como los técnicos que aplican dicha metodología.

Desde sus orígenes se han distinguido diferentes formas de aplicar Investigación-Acción, principalmente dependiendo de las características del proyecto de

investigación [French y Bell, 1996]:

- De diagnóstico:
  El investigador se adentra en una situación problemática, la diagnostica y realiza recomendaciones al grupo crítico de referencia, pero sin que haya un control posterior de sus efectos.
- Participativa:
  El grupo crítico de referencia pone en práctica las recomendaciones realizadas por el investigador, compartiendo con él sus efectos y resultados.
- Empírica:
  El grupo crítico de referencia realiza un registro amplio y sistemático de sus acciones y sus efectos. Esta característica hace que esta variante sea difícilmente aplicable.
- Experimental:
  Consiste en evaluar las diferentes opciones que existen para conseguir un objetivo. El principal inconveniente de esta variante reside en la dificultad de poder medir objetivamente las diversas opciones, ya que por lo general serán, o bien aplicadas en distintas organizaciones con distintas características que enturbian los resultados de la investigación, o bien en una sola organización pero en distintos momentos, con lo que el entorno experimental habrá variado.

Un proceso de investigación que emplea Investigación-Acción se compone generalmente de cuatro actividades organizadas formando un ciclo característico que se ejecuta en forma de espiral, como se muestra en la Figura 1.3. Así, este tipo de proceso de investigación resulta iterativo, dado que se avanza en soluciones cada vez más refinadas mediante la compleción de ciclos, en cada uno de los cuales se desarrollan nuevas ideas, que son puestas en práctica y comprobadas en el ciclo siguiente [Hughes y Seymour-Rolls, 2000; Ruiz y otros, 2002].

- *Planificación*
  Identificar las cuestiones relevantes que guiarán la investigación, que deben estar directamente relacionadas con el objeto que se está investigando y ser susceptibles de encontrarles respuesta. Para ello se buscan caminos alternativos, líneas a seguir o reforzar algo existente. El resultado es la definición de otros problemas o situaciones a tratar.

- *Acción*

  "El plan anterior se pone en práctica".

  Variación de la práctica, cuidadosa, deliberada y controlada. Se efectúa una simulación o prueba de la solución. Es cuando el investigador interviene sobre la realidad.

- *Observación*

  Recoger información, tomar datos, documentar lo que ocurre. Esta información puede tener prácticamente cualquier procedencia (bibliografía, medidas, resultados de pruebas, observaciones, entrevistas, documentos, etc.). También conocido como *evaluación*.

- *Reflexión*

  Compartir y analizar los resultados con el resto de los interesados, de tal manera que se invite al planteamiento de nuevas cuestiones relevantes que proporcionan conocimientos nuevos, y que puedan mejorar las prácticas, modificándolas como parte del propio proceso investigador, para luego volver a investigar sobre estas prácticas una vez modificadas. También conocido como "especificación del aprendizaje" [Wadsworth, 1998]. Algunas variantes de Investigación-Acción no lo consideran como una etapa, sino un proceso continuo que ocurre durante todo el tiempo.

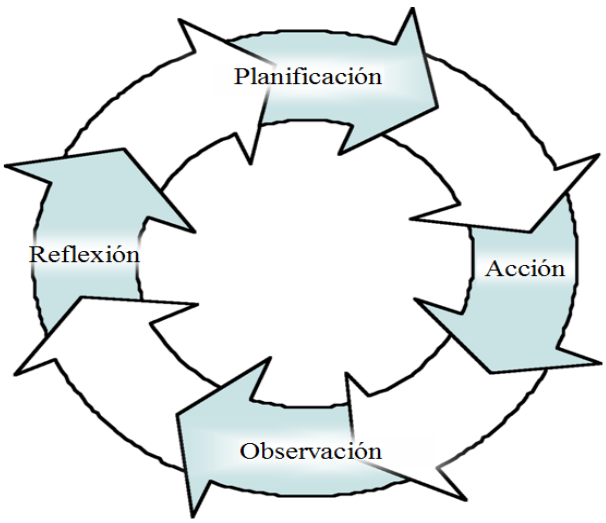


Figura 1.3: Modelo Espiral de actividades del método Investigación-Acción

En los últimos años se ha reconocido la Investigación-Acción como uno de los métodos de investigación (cualitativa) más potentes en el ámbito de los Sistemas de Información, aunque por otro lado, la comunidad de especialistas ha detectado diversos problemas en su aplicación que tienen tres causas fundamentales:

- La falta de método con que los investigadores y practicantes utilizan y conciben IA-SI.
- El contexto de consultoría utilizado, que impone una perspectiva demasiado restrictiva al implicar responsabilidades contractuales e intereses organizacionales que pueden ir en contra de lo propuesto por Investigación-Acción.
- La ausencia de un modelo de proceso de investigación definido que indique los pasos a seguir en IA-SI.

Todo lo anterior puede tener como consecuencia una falta de rigor en el proceso de investigación, para lo cual se ha sugerido que la solución puede hallarse en conducir la investigación con una perspectiva de gestión de proyectos, en la cual se utilice una organización de proyecto conveniente, se incluyan criterios de calidad específicos, y se analicen los factores que inciden en la formalización del proceso. En particular, Estay y Pastor [2000a,b] plantean la necesidad de adoptar prácticas de gestión adecuadas a IA-SI basadas en PMBOK[2]: el modelo de gestión de proyectos (más difundido a nivel internacional) propuesto por el Project Management Institute [PMI, 2000].

Estay y Pastor además consideran que los términos 'Investigación-Acción' y 'proyecto' son conceptos equivalentes, ya que ambos son experiencias de trabajo únicas con resultados finales igualmente únicos, y además comparten la idea de intervención, es decir, ambos suponen una alteración voluntaria de la realidad. Aunque la intervención en Investigación-Acción produce alteraciones en una práctica de trabajo, también es una forma de obtener datos de la experiencia real que son necesarios para el proceso de investigación. Estay y Pastor [2000c] también han propuesto un modelo de madurez basado en el modelo CMM[3] [Paulk y Curtis, 1993], para aplicar prácticas de gestión de proyectos de forma incremental y así garantizar una mejora del rigor y calidad del uso de IA-SI.

En el contexto de la investigación cualitativa en Sistemas de Información se puede considerar que existen dos realidades (científica/académica y práctica) que interactúan

[2] *Guide to the Project Management Body of Knowledge*
[3] por sus siglas en inglés, de *Capability Maturity Model*

pero que se mueven en planos diferentes. IA-SI opera sobre esta realidad dual, que se concreta en dos tipos de ciclos de Investigación-Acción para dos tipos de proyectos:

- Ciclos orientados a resolver problemas dentro de proyectos de Sistemas de Información. Estos proyectos consisten en el desarrollo de una solución informática (son proyectos informáticos, de desarrollo de software, de implantación y/o mantenimiento de sistemas informáticos, etc.). En este caso el investigador se encarga de resolver un problema e Investigación-Acción aparece como una herramienta más para el desarrollo de sistemas de información.
- Ciclos orientados a investigar dentro de proyectos de investigación. Estos proyectos son esfuerzos intencionados buscando un resultado. En este caso Investigación-Acción nos ofrece un método de trabajo y una justificación para acercarnos a una determinada realidad con fines de probar una teoría o hipótesis.

Por otro lado, la estructura de proyecto de IA-SI propuesta por Estay y Pastor [2000b] define dos ciclos característicos:

- Ciclo orientado a construir una solución para generar nuevo conocimiento útil a practicantes y mejorar su práctica. El investigador se conecta con la realidad mediante una intervención. La investigación se utiliza para construir modelos, teorías o conocimiento de manera informada e influida por la realidad. En este ciclo es el interés por resolver un problema lo que origina el interés por la investigación.
- Ciclo orientado a gestionar la investigación para producir nuevo conocimiento a la disciplina de Sistemas de Información y mejorar la práctica de los investigadores. En este ciclo es el interés por la investigación el que origina interés por resolver ciertos problemas.

En resumen, la Investigación-Acción en Sistemas de Información puede analizarse desde dos dimensiones complementarias, tal como se muestra en la Figura 1.4:

- Una dimensión *vertical* en función del tipo de proyecto.
- Una dimensión *horizontal* en función del biciclo típico de la estructura de un proyecto de IA-SI.

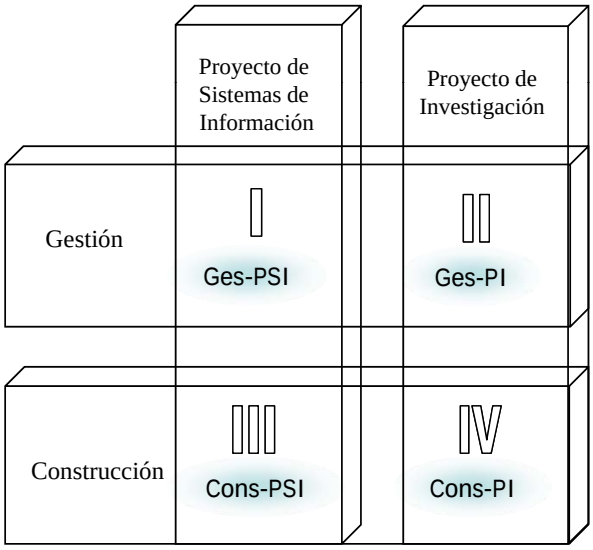


Figura 1.4: Dimensiones de Investigación-Acción en Sistemas de Información

1.3.2. Aplicación de Investigación-Acción

El objetivo principal de la investigación que se ha llevado a cabo en esta tesis es definir un "Proceso de Evaluación sobre la Facilidad de Sustitución de Componentes Software". Por las propiedades de la investigación y los proyectos en los que se enmarca, se aplicará la variante participativa del Método de Investigación-Acción, que como se comentó previamente consta de cuatro roles, de los cuales se definen a continuación sus participantes:

- Investigador:
  El autor de este trabajo, quien es integrante del grupo GIISCo, formado por docentes del Departamento de Ciencias de la Computación de la Universidad Nacional del Comahue, Neuquén, Argentina.
- Objeto investigado:
  Los sistemas basados en componentes, el mantenimiento de los mismos, la actualización de estos sistemas, la posibilidad de reemplazo de componentes, y la aplicación de técnicas de prueba para evaluar la facilidad de sustitución de componentes software.
- Grupo critico de referencia:

Para validar la propuesta se llevarán a cabo diferentes casos de estudio en el ámbito del grupo Alarcos, formado por docentes de la Escuela Superior de Informática de la Universidad de Castilla-La mancha.

- Beneficiarios:
  Aquellas empresas de desarrollo de software, que actualmente son usuarias de componentes software desarrollados previamente en proyectos propios, o bien adquiridos de terceras partes, los cuales se ven afectados por la necesidad de mantener la estabilidad de sus sistemas ante su actualización por medio del reemplazo de componentes software.

1.4. Organización de la Tesis

A continuación se presenta una estructura sintética del resto de los capítulos de esta tesis, indicando el contenido principal de cada uno de ellos.

- **Capítulo 2.** Presenta un panorama del estado del arte en la tecnología de componentes software y su aplicación en diferentes técnicas y enfoques actuales.
- **Capítulo 3.** Introduce nuestra propuesta de un proceso para evaluar la facilidad de sustitución de componentes en sistemas basados en componentes. Además describe el soporte automático del proceso que ha sido desarrollado para alcanzar un enfoque riguroso.
- **Capítulo 4.** Presenta los casos prácticos que se han llevado a cabo para validar la propuesta, y describe con mayor detalle la herramienta automática que da soporte al proceso.
- **Capítulo 5.** Presenta las principales conclusiones y contribuciones del trabajo de esta tesis y describe un análisis de la consecución de los objetivos parciales. Además presenta aspectos destacables como líneas de trabajo futuro.

Capítulo 2

Estado del Arte

2.1. Introducción

En este capítulo se describen los conceptos base para el trabajo de esta tesis, los cuales se relacionan principalmente con el paradigma de Desarrollo del Software basado en Componentes (DSBC), y Pruebas para Componentes Software. Además se expone una comparación de diferentes enfoques significativos, los cuales permiten desarrollar conceptos, opiniones y criterios que facilitan la conceptualización del panorama actual, en el cual se pueden identificar los aspectos que aún deben ser convenientemente abordados. De esta manera resulta factible la comprensión de la propuesta principal de esta tesis que se presenta en el Capítulo 3.

2.1.1. Modelos de Componentes

Un componente OTS es una unidad reutilizable que denota una abstracción simple, encapsulando un conocimiento y comportamiento específicos a través de sus interfaces, y por lo tanto estableciendo una clara separación entre las interfaces y la implementación (encapsulada) del comportamiento del componente [Szyperski, 2002; Gross, 2005; Kung-Kiu y Zheng, 2007].

Esta separación entre interfaces e implementación puede adoptar aspectos variados y distintivos dependiendo de la tecnología utilizada, que se pone de manifiesto en el *Modelo de Componentes* definido, el cual describe las reglas de creación, composición

Categoría	Tipo	Modelo	Lenguaje	
			Definición	Implementación
OO	Clase	EJB	Java	Java
		J2EE		
	Objeto	CCM	OMG IDL	CORBA
		COM	Microsoft IDL	actualmente C, C++ y Ada
		.Net	Microsoft IDL	cualquier lenguaje .Net
		Servicios Web	WSDL	cualquier lenguaje
ADL	Puro	Acme	Acme	– –
		UML2.0	notación UML	– –
	No-Puro	KobrA	notación UML	– –
		Koala	CDL	C
		PECOS	CoCo	C++ o Java
		SOFA	SOFA CDL	Java

Tabla 2.1: Modelos de Componentes

y comunicación de componentes individuales. Los modelos de componentes se pueden clasificar en dos categorías: modelos donde los componentes están basados en orientación a objetos y aquellos donde los componentes son unidades arquitectónicas. Ejemplos de una y otra categoría son EJB (Enterprise Java Beans) [DeMichil y otros, 2001; Monson-Haefel, 2004], y lenguajes de descripción de arquitecturas (ADLs[1]) [Kung-Kiu y Zheng, 2007]. La Tabla 2.1 presenta algunos de los modelos más reconocidos actualmente, los cuales están clasificados de acuerdo a la categorización antes mencionada.

La especificación de un componente, surgida sobre la base del modelo de componentes, puede contener la semántica de sus interfaces, sus restricciones, formatos de datos y protocolos, pero también información detallada de tiempos de espera o de calidad de servicios. Así, la especificación de la interfaz de un componente puede detallar tanto los servicios provistos como los servicios requeridos por el componente, es decir la funcionalidad que realiza el componente y aquella que necesita para exhibir su comportamiento [Crnkovic y otros, 2006; Kung-Kiu y Zheng, 2007].

- *Orientación a Objetos*
  En los Modelos de Componentes basados en OO, el conjunto de métodos que se agrupan en interfaces corresponden a los servicios especificados por el modelo de

[1] del inglés *Architecture Description Language*

componentes. Como resultado, la semántica de este tipo de componentes es una versión mejorada de aquella que corresponde a los objetos. Para la separación entre interfaces e implementación, donde los componentes son en realidad clases Java, el lenguaje de definición coincide con el de implementación, como en el caso de EJB y J2EE (Java 2 Enterprise Edition) [J2EE, 2008]. Por otro lado, para el caso donde los componentes son en realidad objetos, puede existir un lenguaje de descripción de interfaces (IDL[2]) y otro de implementación, como en el caso de COM (Component Object Model) [COM, 2008], CCM (Corba Component Model) [CCM, 2002], la tecnología .Net, y Servicios Web (como se muestra en la Tabla 2.1).

- *Unidades Arquitectónicas*
  En los Modelos de Componentes basados en unidades arquitectónicas, en general los servicios se representan como puertos, los cuales permiten la interconexión entre las unidades arquitectónicas mediante el uso de conectores. Al vincular tales puertos, éstos pueden actuar como servicios provistos y también requeridos, dependiendo del extremo desde donde se observe el vínculo. Con respecto a lenguajes de definición e implementación, en el caso de Acme, UML 2.0 [2003] [Kruchten, 1998] y KobrA sólo existe lenguaje de definición. Mientras que en el caso de Koala, PECOS y SOFA, existen dos lenguajes (como se muestra en la Tabla 2.1).

En los modelos de componentes actuales no existe obligación de enriquecer los componentes con descripciones detalladas de interfaces, restricciones, etc. Dado que el paradigma de DSBC comprende una tecnología inherentemente compleja, podría resultar una carga considerable para los desarrolladores si se los forzara a involucrarse con algún tipo de especificaciones sobre los componentes, lo cual requiere un cierto grado adicional de formalidad y rigor. Por ello es usual que los componentes se provean como código binario con pobres descripciones sobre aspectos de despliegue e integración en formato simple e informal por medio de texto en lenguaje natural. La falta de código fuente y especificaciones completas reduce la aplicabilidad y la efectividad de muchas técnicas clásicas de análisis y pruebas, lo cual se convierte en un desafío para los integradores del sistema y los diseñadores de pruebas [Stuckenholz, 2005; Mariani y otros, 2007].

[2]del inglés *Interface Description Language*

2.1.2. Integración de Componentes

2.1.2.1. Evaluación de Componentes

El DSBC requiere que el contexto propio del sistema (que incluye requisitos, costes, planificación, entornos operativos y de soporte), sea considerado en simultáneo con arquitecturas y diseños factibles, y la disponibilidad y viabilidad de los productos en el mercado. Esto puede hacer más complejo el aseguramiento de calidad de un sistema, dado que los componentes no poseen una calidad conocida y ésta puede además ser afectada por el uso en contextos no previstos.

Se necesita garantizar que la selección de componentes permita cubrir los requisitos especificados, pero además que el proceso de composición (que determina la interrelación de componentes y el mapeo concreto a los requisitos) se encuentre libre de deficiencias. Sin embargo, aún cuando los componentes estuvieran definidos con interfaces claramente documentadas, se confinaran al dominio de origen, y no se combinaran a través de plataformas diversas, se podrían identificar carencias en la cobertura de requisitos o inconsistencias de interacción.

Otro punto de vista implica los objetivos que son de interés para ciertos participantes de un proyecto de desarrollo (es decir *stakeholders*). Estos objetivos equivalen a requisitos funcionales y no-funcionales, los cuales pueden estar en conflicto entre ellos, y con los objetivos de otros stakeholders. Otros conflictos de objetivos involucran: la especificación abstracta de un componente y sus posibles instanciaciones, los objetivos que cumplen los componentes en forma aislada y aquellos que deben cumplir en una composición, y además los aspectos relacionados con los riesgos de adquisición, licencias comerciales, confianza en el vendedor, etc.

La ausencia de código fuente y conocimiento sobre el proceso de desarrollo utilizado para construir los componentes genera un desafío adicional, dado que al construir un sistema usando componentes de diversos vendedores, no se puede tener control directo sobre una gran mayoría de los riesgos asociados con la gestión del proyecto de desarrollo y la operación del sistema.

Existen además diferentes puntos de vista en la observación de una arquitectura software, dado que ésta define los componentes básicos del sistema y sus interconexiones para que se exhiban unas cualidades específicas, y utiliza para ello descripciones de los componentes a distintos niveles de abstracción y con diferente granularidad [Bass y otros,

1998].

La arquitectura de un sistema define los componentes básicos del sistema y sus conexiones, lo cual contribuye a que el sistema exhiba un conjunto específico de cualidades [Bass y otros, 1998]. Una tarea primaria de un ingeniero de sistemas es evaluar las alternativas arquitectónicas que surgen de la forma de priorizar y balancear las cualidades del software, con el fin de alcanzar la mayor eficiencia y efectividad del sistema. Esto conlleva un desafío importante dado que no solamente se requiere evaluar los componentes para identificar si se satisfacen los requisitos funcionales, sino también solucionar convenientemente las contradicciones existentes entre ciertas cualidades (requisitos no-funcionales), como por ejemplo entre rendimiento y mantenibilidad.

En general un componente está descrito en términos de una interfaz que provee acceso a la funcionalidad del componente, y a menudo esta interfaz es la principal fuente de información acerca del componente, dado que la documentación adjunta comúnmente ofrece sólo una guía paso a paso de las operaciones disponibles en su interfaz.

Las características de la interfaz del componente puede influir en el proceso de ensamblaje de componentes y con ello restringir la adopción de ciertas clases de arquitecturas de sistemas. Por ejemplo, la granularidad del acceso a los datos de un componente puede tiene influencia en la forma de llevar a cabo una sincronización de datos entre distintos componentes. Por lo tanto, es necesario identificar todas las posibles fuentes de conflicto en relación a los componentes, sus interfaces y los aspectos que influyen sobre el proceso de integración – que determinarán el grado con el que se satisfacen las cualidades esperadas del sistema [Fioukov y otros, 2002].

La evaluación de un componente podría ser además extendida a la evaluación del proceso de desarrollo usado para crear y mantener un componente. Por ejemplo, asegurar que los algoritmos hayan sido validados y que se hayan llevado a cabo inspecciones rigurosas de código. Actualmente los contratistas (proveedores de servicios y productos software) deberían aplicar prácticas de acuerdo a los estándares de proceso y producto software, y además considerando CMM (el Modelo de Madurez de Capacidad, por sus siglas en ingles) [Torgersson y Dorling, 2002]. De esta manera, se asegura que los componentes software producidos, han sido desarrollados usando prácticas y procedimientos bien definidos, lo cual influye sobre la calidad de sus productos.

2.1.2.2. Adaptación de Componentes

La diversidad en la procedencia de los componentes acarrea una amplia variedad de conflictos, debido a que cada uno en forma aislada posee un conjunto diferente y quizá contradictorio de suposiciones operacionales, pero luego tales componentes deberán participar en una estructura colaborativa que permita la construcción de un nuevo sistema [Brown y Wallnau, 1996].

La probable existencia de tales conflictos exige efectuar un análisis para descubrir y comprender el propósito de cada componente con respecto a su contexto de uso original. El contexto del componente permite aclarar el sentido o alcance de su información, es decir, el significado real detrás de cada aspecto. De esta manera, un componente puede presentar propiedades similares y aún así ser considerado completamente diferente para un nuevo proyecto, al no haber obtenido un resultado favorable en la evaluación de compatibilidad realizada.

Para solucionar algunos de los conflictos, se pueden aplicar técnicas para adaptar los componentes, lo cual a menudo involucra alguna forma de envoltura (*wrapping*), es decir, código que se desarrolla localmente y provee un encapsulamiento del componente para enmascarar el comportamiento no deseado o incompatible. Generalmente el diseño de los wrappers involucra la aplicación de algún patrón de diseño –como aquellos del catálogo de Gamma y otros [1995]– algunos de los cuales permiten independizar a los clientes de alguna funcionalidad específica. A continuación se enumeran algunos tipos de wrappers, pero luego se darán detalles con respecto a otros patrones de diseño y a enfoques que actualmente los aplican [Cechich y otros, 2003; Becker y Schiele, 2003; Gross, 2005; Becker y otros, 2006]:

- *Aislamiento*: provee un único punto de acceso para múltiples interfaces (es decir, una API individual, como por ejemplo ODBC), y se relaciona al patrón de diseño *fachada*;
- *Adaptador*: permite ocultar o adaptar algún aspecto, y se relaciona a los patrones de diseño *adaptador* y *decorador*, entre otros;
- *Instrumentación*: su propósito es instrumentar, depurar, o agregar aserciones, y se relaciona a los patrones de diseño *decorador* y *proxy*, entre otros;
- *Mediador*: informalmente se puede ver como un agente activo que coordina entre diferentes interpretaciones de una misma propiedad del sistema. Por ejemplo,

mediante filtros u otros agentes se pueden convertir o traducir entre diferentes formatos de datos, establecer eventos comunes, o definir políticas administrativas comunes (tales como seguridad y control de acceso). Este wrapper se relaciona a los patrones de diseño *mediador* y *proxy*, entre otros.

Algunas formas de wrappers también se clasifican en base a cuánto acceso se provee a la estructura interna de un componente. Estos enfoques se pueden clasificar como caja blanca, donde el acceso al código fuente permite a un componente ser significativamente reescrito para operar con otros componentes, caja gris; donde el código fuente de un componente no se modifica pero el componente provee su propio lenguaje de extensión o API; y caja negra, donde sólo una forma ejecutable binaria del componente está disponible y no existe ningún lenguaje de extensión o API.

Eliminación de Incompatibilidades Arquitectónicas Asumiendo que se puede efectuar un descubrimiento completo de la interfaz de un componente, el próximo paso es reparar cualquier incompatibilidad que se haya detectado entre los componentes.

Una forma de conducir el descubrimiento de la interfaz es a través de la arquitectura, dado que las arquitecturas software tratan con patrones de diseño de alto nivel, a menudo expresados como componentes, conectores y coordinación. Los componentes se corresponden con las unidades de funcionalidad, los conectores con la integración de los componentes y la coordinación como la manera en la cual los componentes interactúan en ejecución.

Un rol de las arquitecturas software es restringir los tipos de incompatibilidades potenciales que podrían surgir entre las interfaces de los componentes y así ofrecer restricciones útiles para el proceso de descubrimiento de interfaces. Por ejemplo, prescribiendo un modelo de coordinación específico, una arquitectura software en efecto identifica y prioriza las interfaces de ejecución clave que un componente debe exhibir (quizá aplicando algún tipo de wrapper).

En general existe un consenso en que el proceso de adaptación es inherentemente ad-hoc, de poca importancia y algo confuso. Un eufemismo común para una adaptación simple es “pegamento” (*glue*). Otro eufemismo más claro es “wrapper”, que sugiere que un componente es adaptado por medio de encerrarlo dentro de un componente virtual que presenta una interfaz traducida y alternativa.

2.1.2.3. Ensamblado de Componentes

Una vez que se han logrado resolver los conflictos arquitectónicos se pueden ensamblar los componentes mediante alguna infraestructura que provee la conexión entre componentes separados y permita de esta manera formar el nuevo sistema. La definición de una infraestructura se debe observar desde tres niveles:

- En el nivel más alto de abstracción se observa el modelo de coordinación que define cómo los diferentes componentes interactúan para llevar a cabo la funcionalidad. El rol de la infraestructura es permitir la descripción de este modelo de coordinación.
- En el siguiente nivel, se observan los servicios que se proveen para que los componentes interactúen y ejecuten tareas comunes. La interfaz de estos servicios debe ser completa y consistente.
- En el nivel más práctico, se puede considerar la infraestructura como un componente en sí mismo, que implemente los servicios de coordinación requeridos.

Algunas de las infraestructuras más conocidas son: sistemas operativos, gestores de bases de datos, y sistemas middleware. Estos últimos generalmente se utilizan con el fin de construir aplicaciones distribuidas, y algunos ejemplos conocidos tanto para arquitecturas orientadas a componentes, como aquellas orientadas a servicios son: CORBA, Microsoft DCOM y .Net, IBM WebSphere, SUN ONE y J2EE, Borland Visibroker, entre otras [Vinoski, 2003].

2.1.2.4. Actualización de Componentes

De la misma manera que sucede con un sistema tradicional, un sistema basado en componentes debe evolucionar con el tiempo para corregir errores y agregar nueva funcionalidad. Inicialmente un enfoque basado en componentes brinda mayores ventajas en términos de su facilidad de evolución, dado que las unidades de cambio son claramente distinguibles a través de los componentes, y por lo tanto para corregir un error se intercambia el componente defectuoso con un equivalente modificado y corregido. De forma similar, cuando se requiere agregar funcionalidad adicional al sistema, ésta se encapsula en un nuevo componente el cual es adicionado al sistema.

Sin embargo, esta es una vista demasiado simplista de la actualización de componentes. El reemplazo de un componente con otro generalmente implica un gran

esfuerzo y consumo de tiempo, dado que el nuevo componente debe ser cuidadosamente probado tanto en forma aislada como en combinación con el resto del sistema. Esto implica encontrar y corregir todo posible efecto lateral, con el agregado del probable rediseño de los wrappers afectados [Brown y Wallnau, 1996].

Este problema de la actualización de componentes se genera principalmente desde los proveedores de componentes, quienes frecuentemente producen cambios en función de correcciones o mejoras, es decir en base a reportes de error, o bien sobre aspectos competitivos tales como necesidades que se perciben en el mercado y estéticas de los productos. Ante nuevas entregas (*release*) de componentes un integrador de sistemas debe decidir si incluirá o no el nuevo componente en el sistema, dado que se debe enfrentar tanto la potencial re-programación de los wrappers afectados, como todas las pruebas requeridas de los componentes y el sistema. Mientras que si no se efectuaran los reemplazos que se presentan, se deberá confiar en versiones viejas de los componentes, lo cual puede resultar en un atraso con respecto a las nuevas tecnologías, y además perder el soporte tecnológico por parte del proveedor de los componentes.

Como ejemplo del impacto de las actualizaciones de componentes, considérese un sistema que consiste de 12 componentes COTS, de cada uno de los cuales se realiza una nueva entrega (*release*) cada 6 meses. Para mantenerse actualizado con la última versión de cada componente se requiere un promedio de un cambio cada dos semanas. Si las nuevas entregas no se instalan, resulta más y más difícil el análisis de compatibilidad de las distintas versiones de cada componente, generando una pesadilla para el administrador del sistema. Además de todo ello, dado que los sistemas están siendo operados por los usuarios finales, se genera una dependencia que obliga a mantener siempre disponible toda la funcionalidad del sistema [Brown y Wallnau, 1996].

Modificación Predecible de Componentes Los componentes actualizados o modificados han sido reemplazados por versiones más nuevas, o por diferentes componentes con comportamiento e interfaces similares. A menudo se requiere modificar los wrappers desarrollados previamente, y efectuar pruebas para asegurar que el funcionamiento de los componentes que no han cambiado, no se hayan afectado desfavorablemente.

Los sistemas en uso operacional deben periódicamente ser modificados. Típicamente, esto conlleva la implementación de una modificación del sistema, la realización de pruebas

extensivas, para luego efectuar el intercambio con el nuevo sistema. Un número de problemas potenciales pueden surgir al hacer esto:

- identificar y limitar los cambios que requiere el sistema;
- efectuar las suficientes pruebas del sistema para asegurar que las modificaciones no tienen consecuencias no deseables sobre el resto del sistema;

Un sistema basado en componentes sufre de complicaciones en el momento de requerir actualizaciones, dado que los componentes son mantenidos por organizaciones externas. Como resultado, los cambios sobre esos componentes a menudo no son bien entendidos por los integradores del sistema y los usuarios finales, y a menudo no están documentados con suficiente detalle. Esto deja a los integradores de sistemas con el desafío de evolucionar los sistemas basados en componente en una forma predecible, que depende de intentar comprender información incompleta acerca de los componentes bajo modificación.

Sustitución de Componentes Se asume que los componentes deberán ser ensamblados junto con otros componentes para así construir estructuras mas complejas en sistemas de software. Tanto el mecanimso de composición como las técnicas específicas aplicadas para pegar todos los componentes, impone una fuerte influencia sobre la dependencia que tendrán los componentes entre sí. Tal dependencia tiene un impacto significativo sobre el grado de facilidad de sustitución de un componente dentro de un sistema [Mann y otros, 2000]. Cuanto mayor sea la dependencia de un componente respecto de otros componentes en un sistema, más difícil será realizar actualizaciones con nuevas versiones del componente.

Versionado Un esquema común para describir el impacto de los cambios en un producto software es utilizando *números de revisión* con el patrón '*mayor.menor.micro*', donde el número *micro* implica un cambio mínimo que no produce incompatibilidad, el número *menor* ya implica un cambio significativo con un riesgo considerable de incompatibilidad, y el número *mayor* directamente implica un cambio radical generando una completa incompatibilidad. Algunos frameworks de componentes ampliamente conocidos (DCE, CORBA, OSGi [Peterson, 1995; CCM, 2002; The OSGi Alliance, 2005]) y paquetes de desarrollo de sistemas [Debian Project, 2007; Bailey, 1997] han adoptado este esquema.

Aún cuando sea inicialmente útil, no se puede confiar completamente en el esquema de *números de revisión* para asegurar que la estabilidad de un CBS se podrá conservar sin cambios. Un ejemplo simple consiste en cambiar un tipo de dato genérico (por ejemplo *long*) a una versión especializada (por ejemplo *short*) – también conocido como subtipificación [Zaremski y Wing, 1997]. Algunos frameworks de componentes realizan ajustes automáticos sin afectar el sistema, sin embargo en plataformas con verificación automática de tipos y enlace (*binding*) como la máquina virtual Java (JVM), tal cambio mínimo puede causar una excepción debido a la definición de tipo que espera el código compilado de un cliente (es decir, *long*). La razón de ello es que JVM no puede adaptar dinámicamente la invocación enlazada estáticamente a la nueva versión del servicio con un subtipo teóricamente seguro [Brada y Valenta, 2006].

Componentes Java y .Net En el lenguaje Java un *paquete* puede ser considerado como la unidad de despliegue, y por lo tanto siendo visto como un componente. Java mapea clases a ficheros simples y paquetes a directorios en un sistema de ficheros. A diferencia del término paquete en UML, un paquete Java es una organización física de clases, recursos y un manifiesto. Dado que los ficheros binarios de Java contienen meta información, se pueden utilizar mecanismos de reflección para obtener en tiempo de ejecución información de los tipos e interfaces exportadas por los componentes [Stuckenholz, 2005]. Además los *paquetes* son las unidades de versionado en Java, donde el manifiesto del paquete puede opcionalmente ser usado para enriquecer un componente con información acerca de versionado, incluyendo el nombre del componente, el número de versión, el nombre de la especificación y la versión de la especificación [Riggs, 1998].

El framework .Net también provee metadatos en el modelo de componente. Sus recursos tales como clases, ejecutables y componentes se pueden también desarrollar en los llamados *ensambles (assemblies)* en conjunto con un manifiesto. Por lo tanto los *ensambles* son las unidades de versionado, donde el manifiesto puede contener el número de versión (un cuádruplo de enteros de 16 bits), y algunos metadatos como el nombre del ensamble e información del fabricante, que puede ser recuperado en tiempo de ejecución por medio de reflección [Stuckenholz, 2005].

Sin embargo, los desarrolladores de los componentes no están forzados a enriquecer los componente con tales metadatos y esto requiere establecer una política rigurosa de versionado para que los metadatos resulten realmente útiles.

En Java, los usuarios de los componentes son los responsables de evaluar la información de versionado por si mismos. La clase abstracta *Package* contiene un método denominado *isCompatibleWith* el cual se supone puede recibir la versión de una especificación y verificar la compatibilidad del componente actual con la versión dada. Aún cuando los desarrolladores provean una vaga noción de compatibilidad en tal método, dado que Java no provee herramientas automáticas para realizar tales cálculos, los desarrolladores pueden arrivar a suposiciones erróneas que causen fallas de integración en los sistemas actuales [Stuckenholz, 2005].

El sistema de versionado del framework .Net es actualmente el mecanismo más progresivo, aunque esto depende fuertemente de los metadatos que se insertan en los ensambles, especialmente el número de versión de los componentes. Dado que los desarrolladores necesitan especificar los metadatos manualmente, se pueden generar algunos efectos impredecibles cuando los cambios no son reflejados correctalmente en los números de versión [Stuckenholz, 2005].

2.2. Pruebas para Componentes Software

En la sección previa se discutieron los desafíos que debe enfrentar un ingeniero de software ante la utilización de componentes para solucionar el conjunto de requisitos funcionales y no funcionales de un sistema. Sin embargo existe una consideración adicional en relación a la confianza que puede tener un ingeniero de software sobre un componente candidato, lo cual surge directamente de la fiabilidad de los servicios que ofrece el componente [Mariani y otros, 2007; Jaffar-UrRehman y otros, 2007]. Es decir además del desafío de superar los conflictos de integración, un ingeniero de software debe tener en consideración la probabilidad de que los componentes adquiridos sufran algún tipo de fluctuación inesperada en su funcionamiento, producto de una diferencia en el perfil de uso del componente con respecto a los otros componentes en el sistema de destino, o simplemente porque no se encuentra totalmente libre de defectos (*fault-free*) [Gross, 2005; Mariani y otros, 2007]. La solución a esta problemática está fuertemente ligada a la capacidad de validación de ese componente (cuán adecuado es con respecto al propósito de uso) [Bertolino y Polini, 2003].

Estos aspectos tienen su explicación desde la perspectiva de las técnicas de Pruebas del Software, que de acuerdo a SWEBOK [2004] consiste de la verificación dinámica

del comportamiento de un programa contra el comportamiento esperado, por medio de un conjunto finito de casos de prueba, que fueron adecuadamente seleccionados de un conjunto usualmente infinito de ejecuciones en un dominio. Existen diferentes estrategias de prueba que dependen de varios factores tales como la disponibilidad de especificaciones o código fuente, pero también del tipo de actividad de prueba que se desarrolla. Estas actividades se pueden clasificar de acuerdo a su propósito en distintas fases de prueba, que a continuación se explican inicialmente en el contexto del software procedural y orientado a objetos (OO), dado que no solamente han servido de base para la evolución hacia el DSBC, sino que en particular los componentes software se desarrollan generalmente en función del paradigma OO [Vincenzi y otros, 2003; Jaffar-UrRehman y otros, 2007].

2.2.1. Prueba Tradicional

De manera similar a un proceso de desarrollo que está dividido en varias fases, permitiendo que el ingeniero de software implemente su solución paso a paso, la actividad de pruebas también se divide en distintas fases. El responsable de pruebas puede de esta manera concentrarse en diferentes aspectos del software y usar diferentes criterios de prueba en cada fase.

2.2.1.1. Fases de Prueba Tradicional

La actividad de pruebas puede dividirse en general en tres fases incrementales de acuerdo al paradigma procedural: pruebas de unidad, integración y sistema; aunque en el caso de la orientación a objetos (OO) existen ciertas variaciones que se presentan en forma comparativa en la Tabla 2.2 y se explican a continuación [Pressman, 2000; Vincenzi y otros, 2003]. Además se puede considerar una fase adicional: la prueba de regresión, que se aplica cada vez que alguna pieza de software ha sufrido alguna modificación considerable, la cual adquiere su mayor relevancia al ingresar a la etapa de mantenimiento del software.

Prueba de Unidad Las pruebas de unidad están enfocadas en unidades individuales, y de acuerdo al Glosario Estándar IEEE 610 [1990], una unidad es un componente de software que no puede ser dividido. El objetivo de esta fase es asegurar que los aspectos algorítmicos estén correctamente implementados, para lo cual se identifican errores relacionados con la lógica e implementación de cada unidad [Vincenzi y otros, 2003].

Fase	Prueba Procedural	Prueba Orientada a Objetos	
		Menor Unidad: Método	Menor Unidad: Clase
Unidad	Intra-procedural	Intra-método	Intra-método, Inter-método e Intra-clase
Integración	Inter-procedural	Inter-método, Intra-clase e Inter-clase	Inter-clase
Sistema	Sistema completo	Sistema completo	Sistema completo

Tabla 2.2: Relación entre Fases de Prueba Procedural y OO

En programas procedurales una unidad corresponde a una subrutina o procedimiento, y las pruebas de unidad también se denominan *intra-procedural*. En OO la menor parte a ser probada es un método, y las pruebas de unidad se denominan *intra-método* [Harrold y Rothermel, 1994].

Prueba de Integración Luego de que cada unidad ha sido probada, se da inicio a la integración de las unidades y en consecuencia a la fase de prueba de integración, que resulta imprescindible dada las limitaciones que presenta la prueba de unidad para asegurar que cada unidad funcione en cada uno los distintos escenarios de interacción con otras unidades [Vincenzi y otros, 2003].

Dado que los métodos de una misma clase en general interactúan para implementar una cierta funcionalidad, se podría considerar una clase como la integración de métodos que correspondería a pruebas de integración denominadas *inter-método*. Por otro lado, en el paradigma procedural las pruebas de integración también se denominan *inter-procedural* [Harrold y Rothermel, 1994].

Se han definido además otro dos tipos de pruebas OO: *intra-clase* e *inter-clase*. En pruebas *intra-clase* se prueban las interacciones entre métodos públicos a través de diferentes secuencias de invocaciones a tales métodos, con el objetivo de identificar probables secuencias que lleven a un objeto a un estado inválido. Dado que el usuario puede invocar los métodos públicos en un orden diferente cada vez, este tipo de pruebas brindan una cierta confianza de que las diferentes secuencias de invocaciones actuarán correctamente. En pruebas *inter-clase* se aplica el mismo concepto a métodos públicos pero no solamente para aquellos pertenecientes a una única clase, es decir, las pruebas requieren invocaciones entre métodos en diferentes clases [Harrold y Rothermel, 1994].

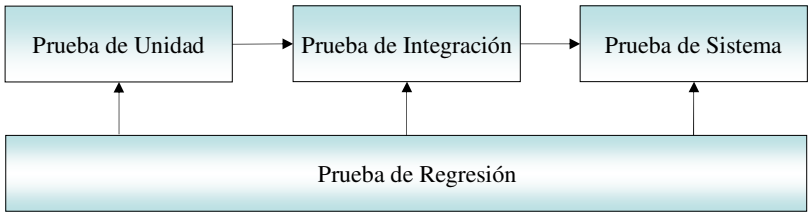


Figura 2.1: Fases de Prueba en Desarrollo Tradicional

Algunos autores consideran que la menor unidad en OO es una clase [Binder, 2000; Vincenzi y otros, 2003], por lo cual la prueba de unidad estaría compuesta de pruebas *intra-método*, *inter-método*, e *intra-clase*, mientras que la prueba de integración correspondería a pruebas *inter-clase*.

Prueba de Sistema Una vez que el software ha sido integrado y trabaja como un todo, debería remitirse a la pruebas de sistema, cuyo objetivo es asegurar que el software y los elementos anexos del sistema (hardware y base de datos, entre otros) han sido adecuadamente combinados y se han logrado la funciones y rendimiento esperados. En esta fase se aplican estrategias funcionales, y no existe una gran diferencia entre software OO y procedural [Pressman, 2000; Vincenzi y otros, 2003].

Prueba de Regresión Cuando una unidad ha pasado exitosamente las pruebas realizadas, puede requerir algún tipo de cambio, que en general se relaciona con las estructuras de datos utilizadas, la mejora de sus algoritmos, una necesidad de adaptación a nuevos entornos, o bien aspectos sociales del contexto de uso (por ejemplo nuevas políticas comerciales). Ante cualquier modificación, tanto la unidad directamente afectada como el sistema completo requieren que se vuelva a verificar la corrección comprobada anteriormente y que se garanticen los requerimientos no-funcionales alcanzados, como se observa en la Figura 2.1. Esto lleva por lo tanto a repetir las pruebas realizadas para verificar que el comportamiento del software no ha cambiado, excepto en las partes que así lo requerían [Bertolino y Marchetti, 2005].

Se puede definir la prueba de regresión como la repetición selectiva de un conjunto de prueba (TS, de *test suite*) previamente ejercitado sobre un sistema o unidad, para comprobar que las modificaciones no hayan causado efectos laterales [IEEE 610, 1990; SWEBOK, 2004; Veenendaal, 2006]. En la práctica, la idea es mostrar que la versión

previa del software (n) que pasó exitosamente las pruebas, puede volver a hacerlo luego de la modificación sufrida (que ha generado una nueva versión $n + 1$). Pero además surge la inquietud acerca de cuál es el volumen requerido de pruebas de la versión $n + 1$ utilizando las pruebas de la versión n [Whittaker, 2000].

Es decir, el esfuerzo que insume la actividad de pruebas del software genera una controversia entre la potencial garantía dada por la prueba de regresión y los recursos requeridos para su consecución en cada oportunidad que se efectúa un cambio en el software. Dados los costes involucrados generalmente se aplican estrategias de selección o priorización de casos de prueba para así obtener un TS reducido que mejore la eficiencia de la prueba de regresión, sin perder seguridad (es decir, que aun se puedan exponer fallos en las piezas esperadas) [Rothermel y Harrold, 1996; Bertolino y Marchetti, 2005; Orso y otros, 2007].

2.2.1.2. Estrategias de Prueba Tradicional

A continuación se realiza una revisión de las estrategias de prueba tradicional y su potencial uso en el contexto del software basado en componentes. En general las estrategias en el software tradicional se clasifican en pruebas "*basadas en especificación*" y pruebas "*basadas en código*" [Binder, 2000; Vincenzi y otros, 2003; Bertolino y Marchetti, 2005]. En la Tabla 2.3 se puede observar el uso de tales estrategias en las distintas fases de prueba tradicional:

- *Pruebas basadas en Especificación*

  También denominadas pruebas de "*caja negra*", encontramos técnicas *funcionales* y las *basadas en estado*. El objetivo es revelar defectos (*faults*) relacionados con la funcionalidad externa, a la comunicación de las interfaces entre módulos, a las restricciones requeridas (pre- y post-condiciones), y al comportamiento de un programa. El problema es que generalmente la especificación existente no es formal, lo cual dificulta la creación de un TS para ejercitar los programas en forma sistemática. Sin embargo, los criterios basados en especificación pueden usarse en cualquier contexto (procedural y OO) y en cualquier fase sin necesidad de una adaptación importante.

- *Pruebas basadas en Código*

También denominadas pruebas de "*caja blanca*", encontramos técnicas *estructurales* y las *basadas en defectos* (*fault-based*). El objetivo es identificar errores en la estructura interna y en el comportamiento de los programas, y para ello, requieren el manejo del código fuente y la selección de un TS que ejercite piezas específicas del código, no su especificación. La desventaja es que este enfoque puede depender de varios factores tales como el lenguaje de programación y la necesidad de tener acceso al código fuente. Este factor en el caso de componentes OTS, por ejemplo, no siempre es posible.

Tipo	**Estrategia**	**Fase**†			
		U	**I**	**S**	**R**
basada en Especificación	Funcional	√	√	√	√
	basada en Estado	√	√	√	√
basada en Código	Estructural	√	√		√
	basada en Defectos	√	√		√

† Fases de Prueba: Unidad (U), Integración (I), Sistema (S), Regresión (R)

Tabla 2.3: Estrategias y Fases de Prueba Tradicional

Prueba Funcional Se utiliza la especificación para obtener los requerimientos de prueba o los datos de prueba, sin ningún conocimiento sobre la implementación [Pressman, 2000]. Se requiere una especificación de alta calidad que se corresponda con los requerimientos del cliente, para así soportar la aplicación de los criterios funcionales. Ejemplos de tales criterios son [Binder, 2000]: 1) Partición de Equivalencia; 2) Análisis de Valores Límite; 3) Grafo Causa-Efecto; y 4) Partición por Categoría [Ammann y Offutt, 1994].

Con el uso de las técnicas funcionales puede resultar difícil cuantificar la actividad de prueba, dado que no siempre es posible asegurar que ciertas partes esenciales de la implementación del producto hayan sido ejercitadas. Otro problema es que las especificaciones (no formales) pueden estar incompletas o ser inconsistentes y por lo tanto de la misma manera resultará el conjunto de prueba que se genere en base a ellas. No obstante, dado que los criterios funcionales están basados sólo en la especificación, se pueden utilizar tanto para las pruebas procedurales, como para las pruebas OO, e incluso para las pruebas en componentes software [Vincenzi y otros, 2003; Bertolino y Marchetti, 2005].

Prueba Estructural Se utilizan los aspectos de implementación para determinar los requerimientos de prueba. La mayoría de los criterios de la técnica estructural se basan en un Grafo de Flujo de Control (también denominado Grafo de Programa) para representar el programa bajo prueba.

Los primeros criterios estructurales se basaron exclusivamente en estructuras de flujo de control, tales como el *criterio de nodos* (sentencias o comandos), el *criterio de arcos* (o decisiones), y el *criterio de los caminos* [Pressman, 2000; Myers, 2004]. Luego aparecieron los criterios basados en flujo de datos que exponen las interacciones entre las definiciones y los usos de las variables, dado que aun para programas pequeños, las pruebas basadas en flujo de control no son efectivas para revelar ciertos errores triviales. Los criterios de flujo de datos proveen una jerarquía de criterios desde el *criterio de arcos* al *criterio de los caminos*, tratando de que las pruebas sean una actividad más rigurosa. Entre los mejores criterios de flujo de datos conocidos se encuentran aquellos introducidos por Rapps y Weyuker [1982], tales como, el *criterio de los p-usos*, *de las definiciones*, *de los usos*, y *de los du-caminos*.

Se han generado diversas extensiones de los criterios de flujo de datos, tanto para pruebas de integración de programas procedurales, como de pruebas de unidad e integración para OO [Harrold y otros, 1999; Binder, 2000].

Las pruebas estructurales enfrentan serias restricciones y desventajas en relación con la necesidad de determinar requerimientos de prueba inviables tales como caminos no ejecutables y asociaciones imposibles. Estas restricciones generan serios problemas para la automatización de las pruebas. No obstante, esta técnica parece ser complementaria a la prueba funcional y la información que se obtiene con su aplicación también resulta relevante para el mantenimiento, depuración y estimación de fiabilidad del software [Pressman, 2000; Vincenzi y otros, 2003; Bertolino y Marchetti, 2005].

Prueba basada en Defecto Se utiliza información de los defectos que se encuentran con mayor frecuencia en los distintos proyectos de desarrollo y además considera los tipos de defectos específicos que un ingeniero de pruebas desea descubrir. Existen dos criterios que típicamente se concentran en defectos: Inyección de Errores (*error seeding*) y Análisis Mutacional [DeMillo y otros, 1978; Vincenzi y otros, 2003; SWEBOK, 2004].

En la técnica de Inyección de Errores se introducen en el programa bajo prueba un número conocido de defectos artificiales previo a ejercitar la prueba, para que luego de la

prueba se calcule el ratio entre defectos naturales/artificales a partir del número total de defectos encontrados, y con ello se estima el número de defectos naturales restantes. Los problemas con este enfoque son 1) los defectos artificiales pueden ocultar a los defectos naturales; 2) para obtener un resultado confiable estáticamente es necesario usar programas que tengan 10000 defectos o más; y 3) se basa en la suposición que los defectos se distribuyen uniformemente en el programa, aunque en programas reales se observan grandes piezas de código simple con pocos defectos y pequeñas piezas con una alta complejidad y alta concentración de errores [Budd, 1981; DaSilva y otros, 2007].

El Análisis Mutacional se basa en aplicar ligeros cambios al código de un programa bajo prueba (PUT[3]), generando nuevas versiones que deberían comportarse de forma defectuosa. Las versiones defectuosas se denominan mutantes, y se crean en base a operadores de mutación, que son reglas que definen los cambios sintácticos que se aplican sobre el PUT para crear los mutantes. El propósito es evaluar la capacidad de un TS para distinguir las diferencias entre PUT y sus mutantes. Cuando un mutante tiene un comportamiento diferente al de PUT (es decir, las salidas de PUT y del mutante son diferentes para algún caso de prueba de TS) se dice que el mutante "muere"; si no, el mutante permanece "vivo". Un mutante vivo debe ser analizado para comprobar si es funcionalmente equivalente al PUT o si puede crearse algún nuevo caso de prueba que permita matar al mutante, promoviendo de esta manera la mejora del TS. Uno de los problemas con la técnica de mutación es el alto coste de ejecutar un gran número de mutantes [DeMillo y otros, 1978; Choi y otros, 1989; Duncan, 1993; Andrews y otros, 2005].

Se han propuesto extensiones de este criterio para pruebas de integración y para especificaciones de programas. Delamaro y otros [2001] han definido el criterio de Mutación de Interfaces que aplica el concepto de mutación a la fase de prueba de integración. Con este criterio se propuso un nuevo conjunto de operadores mutantes especializados en errores de integración [Gosh y Mathur, 2001].

En el contexto de las especificaciones de prueba, la mutación puede ser usada para las pruebas de Redes de Petri [Fabbri y otros, 1995], y Máquinas de Estado Finito (FSM[4]) [Vincenzi y otros, 2003], entre otros.

Además se han propuesto enfoques para OO, que no difieren significativamente

[3]del ingles *Program under Test*
[4]del ingles *Finite State Machines*

de los operadores mutantes tradicionales, pero introducen operadores especializados correspondientes a aspectos propios de la OO tales como clases y relaciones entre clases [Kim y otros, 2001; Ma y otros, 2005; Offut, 1995; Offutt y otros, 1996; Smith y Williams, 2007a,b].

Prueba basada en Estado Se utiliza una representación basada en estados de la unidad o componente bajo prueba. Basado en este modelo, los criterios para generar secuencias de prueba se utilizan para asegurar un comportamiento correcto [Chow, 1978; Sabnani y Dahbura, 1988; Fujiwara y otros, 1991]. Como se mencionó previamente, las técnicas de mutación también han sido aplicadas en el caso de FSM [Vincenzi y otros, 2003].

Los criterios basados en FSM también son ampliamente usados en el contexto de OO para representar los aspectos de comportamiento de los objetos. Entre los criterios más conocidos se destacan el *criterio de estados*, *de eventos* y *de acciones* entre los más débiles, luego el *criterio de transiciones* que cubre a los anteriores [Ball y otros, 2000; Hong y otros, 2001; Grieskamp y otros, 2001; Offutt y otros, 2003], y luego criterios de cobertura más fuertes como el *criterio de caminos ida-vuelta* particularmente propuesto por Binder [2000].

También se utilizan los criterios de FSM en el contexto del software basado en componentes dado que sólo requieren una representación basada en estado para ser aplicados [Beydeda y Gruhn, 2001].

2.2.2. Perspectivas de Prueba para Componentes

En el contexto de DSBC se debe considerar la actividad de prueba para componentes software desde dos puntos de vista, de acuerdo a Harrold y otros [1999]: el rol del *usuario* y el rol del *proveedor* del componente, los cuales se explican a continuación y se resumen en la Tabla 2.4.

Usuario del Componente: Como explica la Tabla 2.4, los usuarios son quienes desarrollan los sistemas por medio de la integración de componentes OTS, para los cuales se han propuesto algunos enfoques que en realidad adaptan técnicas tradicionales de análisis y pruebas para ser aplicados en DSBC. Sin embargo, dado que el código fuente

Proveedor de Componente	Usuario de Componente
Desarrolla el componente	Usa el componente
Código fuente disponible	Código Fuente no disponible
Pruebas de caja blanca y caja negra	Pruebas de caja negra
Vista del componente independiente del contexto	Vista del componente dependiente del contexto
Se deben poner bajo prueba todas las configuraciones y aspectos de comportamiento del componente	Se prueba sólo un subconjunto de las configuraciones y aspectos de comportamiento del componente que están relacionadas a la aplicación de destino
Prueba de Unidad para asegurar la correctitud de la funcionalidad	Prueba de Integración para invocar los servicios del componente
Bajo coste de corrección de errores	Alto coste de corrección de errores

Tabla 2.4: Perspectivas de Pruebas para Componentes Software

no siempre es accesible, se restringe la aplicabilidad de ciertas técnicas y criterios de cobertura de caja blanca tales como análisis de flujo de datos [Rapps y Weyuker, 1982; Binder, 2000] y mutación [DeMillo y otros, 1978; Offut, 1995], para los cuales se debe considerar alguna alternativa viable, como por ejemplo mutación de interfaces [Gosh y Mathur, 2001; Delamaro y otros, 2001]. Aun si el código estuviera disponible, los componentes y el sistema basado en componentes pueden haber sido desarrollados en lenguajes diferentes, por lo cual una herramienta que analiza y prueba el sistema completo podría fallar en analizar ciertos componentes [Vincenzi y otros, 2003].

Es frecuente que un componente software ofrezca más funcionalidad que aquella que el cliente necesita, con lo cual sería necesario identificar la pieza de funcionalidad que realmente requiere el usuario, de lo contrario una herramienta de pruebas no podrá suministrar reportes útiles. Por ejemplo, los criterios estructurales evalúan cuánto de los elementos requeridos son cubiertos por un TS, pero en un sistema basado en componentes la parte no usada de los componentes debería ser excluida de esta evaluación, si no, la herramienta podría determinar que el cubrimiento evaluado es bajo, aun si el TS fuera bueno para la porción de código usada [Rosenblum, 1997; Briand y otros, 2006; Jaffar-UrRehman y otros, 2007].

Desarrollador del Componente: El proveedor del componente implementa y prueba el componente independientemente de cuál será la aplicación de destino. A diferencia

del usuario, el proveedor tiene acceso al código fuente, por lo tanto las pruebas del componente utilizan las estrategias tradicionales de pruebas de unidad e integración (como resume la Tabla 2.4). Sin embargo, los criterios tradicionales tales como los basados en flujo de control pueden no ser suficientes para la prueba de los componentes debido a que son ineficientes para revelar los fallos más significativos [Vincenzi y otros, 2003]. Corregir un fallo en un componente luego de su entrega (*release*) tiene un alto coste, mucho más alto que si el fallo se encontrara durante las pruebas de integración en sistemas tradicionales, dado que el componente será probablemente usado en un gran número de aplicaciones.

El proveedor debe tener los mecanismos para realizar pruebas efectivas del componente como una unidad de software independiente. Al hacer esto, el proveedor incrementa la confianza del usuario en la calidad del componente y reduce los costes de prueba para el usuario [Jaffar-UrRehman y otros, 2007]. Rosenblum [1997] describe un enfoque para pruebas de unidad de componentes que depende del contexto de aplicación y de esta manera resulta más relevante para el usuario que para el proveedor.

2.2.2.1. Facilidad de Prueba de Componentes Software

De acuerdo al Glosario Estándar IEEE 610 [1990], la Facilidad de Prueba del Software considera dos aspectos: 1) el grado en el cual un sistema o componente facilita que se establezcan criterios de prueba y el rendimiento de las pruebas para determinar si esos criterios han sido cubiertos; 2) el grado en el cual un requerimiento es establecido en términos de permitir que se establezcan criterios y rendimiento de pruebas para determinar si esos criterios han sido cubiertos.

En la definición previa se realizan dos consideraciones: la forma en que se desarrolla el software y los requerimientos sobre el sistema software que sirven de base para la definición de un TS. En la industria del software se requiere de componentes reutilizables que puedan ser efectivamente probados, para así satisfacer la gran demanda de desarrollos eficaces en reducción de tiempo y costes. Para esto, los proveedores de componentes necesitan disponer de guías y métodos prácticos y válidos para desarrollar componentes software con capacidad de ser probados [Jaffar-UrRehman y otros, 2007].

En particular para la Facilidad de Prueba en DSBC se ha establecido un conjunto de métricas que son esenciales para que los proveedores puedan evaluar el nivel de facilidad de prueba de los componentes, y los usuarios puedan realizar selección y

evaluación de componentes y determinar el esfuerzo que ha invertido el desarrollador del componente para incrementar su capacidad de ser probado. A continuación se listan las métricas de Facilidad de Prueba para Componentes Software, las cuales se resumen en la Tabla 2.5 [Freedman, 1991; Jaffar-UrRehman y otros, 2007].

Métrica	**Descripción**
Facilidad de Comprensión	Información anexa sobre el componente (ejemplo: documentación de las interfaces, de pruebas, etc.)
Facilidad de Observación	Percibir los datos de *salida* como función de las *entradas*
Facilidad de Control	Producir datos de *salida* a partir de un conjunto específico de datos de *entrada*
Facilidad de Trazado	Comparación del comportamiento esperado con la ejecución del componente (caja negra). Comprobación del estado interno del componente en cada invocación a sus interfaces (caja blanca)
Capacidad de Soporte de Prueba	Mecanismos para agilizar las pruebas de componentes (ejemplo: para generar y ejecutar un TS, para administrar el proceso de prueba, etc)

Tabla 2.5: Métricas de Facilidad de Prueba para Componentes Software

- *Facilidad de Comprensión del Componente* [Jaffar-UrRehman y otros, 2007]

  Las descripciones de las interfaces del componente que explican su funcionalidad permiten mejorar la comprensión sobre el componente y facilitan la reutilización (el mayor beneficio de DSBC). Además cualquier otra información que se adjunta sobre el componente provee un medio para mejorar la comprensión sobre el componente.

- *Facilidad de Observación del Componente* [Freedman, 1991]

  Un componente software es observable si las distintas salidas son generadas de distintas entradas de manera que, si se repite una entrada de prueba, la salida es la misma. Si las salidas no son las mismas, entonces el componente es dependiente de estados ocultos que son internos al componente. Por lo tanto, el mapeo de las entradas con sus correspondientes conjuntos de salida para cada interfaz del componente puede evaluar la observación del componente. Si el comportamiento del componente puede cambiar internamente, entonces requiere que se determinen todas las posibles salidas para una entrada particular. Tal información para cada una de las interfaces del componente mejora la observación del mismo.

- *Facilidad de Control del Componente* [Freedman, 1991]

  El grado de control sobre el componente está en relación con la facilidad de producir una salida específica a partir de una entrada específica, es decir, se refiere al mapeo de una salida del componente a un conjunto de entradas específicas. Al comprobar la ejecución normal de un componente software con entradas definidas se puede medir el control sobre el componente para producir las salidas esperadas definidas.

- *Facilidad de Trazado del Componente* [Gao y Wu, 2003]

  El trazado de un componente software se clasifica en caja negra y caja blanca. La facilidad de trazado de caja negra puede ser evaluada comparando el comportamiento esperado del componente con los resultados de la ejecución del mismo, mientras que el trazado de caja blanca se establece comprobando el estado interno de los componentes en cada invocación a sus interfaces. Esta métrica involucra varios tipos de trazas: operacional, eventos, estado, rendimiento (*performance*) y errores.

- *Capacidad de Soporte de Pruebas del Componente* [Jaffar-UrRehman y otros, 2007]

  La capacidad de soporte de pruebas de componentes involucra mecanismos que agilicen las pruebas, es decir, herramientas que automaticen las distintas tareas del proceso de pruebas. Por ejemplo: para generar y ejecutar un TS, para administrar el proceso de pruebas (que involucra planes y documentación, entre otros), etc.

2.2.2.2. Criterios de Cobertura para Componentes

Como se mencionó previamente el proveedor tiene acceso al código fuente de los componentes que desarrolla y por lo tanto puede aplicar distintos métodos y estrategias de caja blanca. Sin embargo los usuarios que no disponen del código fuente de los componentes, requieren de criterios de cobertura de prueba específicos que les permitan evaluar la correctitud del componente como una unidad que se integra a la aplicación de destino. A continuación se introducen algunas definiciones de los criterios de cobertura para componentes [Jaffar-UrRehman y otros, 2007].

- *Criterio de Métodos* [Gosh y Mathur, 2001].

  Los componentes pueden tener varias interfaces, cada una compuesta de un conjunto de métodos o servicios. El criterio requiere que toda interfaz deba ser

ejecutada al menos una vez. Esto significa que todo servicio de cada interfaz debe ser invocado al menos una vez. Este criterio de cobertura también ha sido denominado *criterio de interfaces* [Wu y otros, 2001].

- *Criterio de Eventos* [Wu y otros, 2001].

  Un evento es un incidente en el cual el efecto resultante es la invocación de una interfaz. Los eventos pueden ser síncronos (por ejemplo, invocaciones directas a los servicios) o asíncronos (por ejemplo, la ocurrencia de excepciones). El criterio requiere que todo evento (síncrono o asíncrono) de un componente deba ser cubierto por algún caso de prueba. Así este criterio cubre a uno más específico denominado *criterio de excepciones* [Gosh y Mathur, 2001; Wu y otros, 2000].

- *Criterio de Dependencia de Contexto* [Wu y otros, 2001].

  Los eventos pueden tener dependencias secuenciales con otros eventos, causando distintos comportamientos de acuerdo al orden en el que son invocados (servicios o excepciones). El criterio requiere que cada una de las secuencias operacionales sea atravesada al menos una vez.

- *Criterio de Dependencia de Contenido* [Wu y otros, 2001].

  Una interfaz puede efectuar cambios en ciertos valores que afecten el comportamiento de otra interfaz. Este cubrimiento corresponde a una forma de estrategia de análisis de flujo de datos.

Los criterios presentados previamente describen un ordenamiento de acuerdo a la relación de subsumción, de menor a mayor cobertura. En el caso de los últimos dos criterios se pueden considerar dos casos particulares: dependencia *intra-* o *inter-componente* de interfaz, de acuerdo a lo siguiente [Wu y otros, 2000].

a) *Criterio de Dependencia de Contexto*

- *Intra-componente*. Cuando los *eventos internos* a la interfaz del componente presentan interdependencia entre sí,
- *Inter-componente*. Cuando los eventos de la interfaz de un componente presentan dependencia con *eventos externos*. Los eventos externos se corresponden con eventos provocados por un componente y atendidos por otro, lo cual genera interacciones entre dos componentes. Esto requiere diseñar

pruebas en las cuales interviene un componente cliente y un componente servidor.

b) *Criterio de Dependencia de Contenido*

- *Intra-componente*. Cuando los servicios miembros de una interfaz de un componente presentan interdependencia entre sí,
- *Inter-componente*. Cuando los servicios de una interfaz de un componente presentan dependencia con servicios de un componente diferente. Esto requiere diseñar pruebas en las cuales interviene un componente cliente y un componente servidor.

2.2.3. Técnicas de Prueba para Componentes

De acuerdo a lo expuesto previamente se puede observar que existe una fuerte relación entre las fases y estrategias tradicionales con aquellas para pruebas en el contexto de DSBC, sin embargo también surgen consideraciones particulares en relación con la perspectiva proveedor/usuario de los componentes y la capacidad de prueba que presentan los mismos. A continuación se explican las fases de prueba para componentes indicando la posibilidad de uso (con mayor o menor ajuste) de estrategias tradicionales, y el desarrollo de nuevas estrategias específicas para pruebas en el contexto de DSBC.

2.2.3.1. Fases de Prueba para Componentes

Al analizar las fases de prueba en el contexto de DSBC se deben reconsiderar las actividades involucradas, como así también quiénes son los responsables de las mismas. En este sentido, Polini y Bertolino [2005] proponen una adaptación de las fases tradicionales de prueba, según se muestra en la Figura 2.2, en la cual se identifican las extensiones aplicadas y los roles asociados con cada fase. A continuación se explican las fases de prueba para componentes software.

Prueba de Unidad La menor unidad a ser probada está representada por un componente, y así esta fase también se denomina *prueba de componente*. La prueba de componentes realizada por el desarrollador del componente tiene el objetivo de establecer el funcionamiento apropiado del componente y detectar posibles fallos en

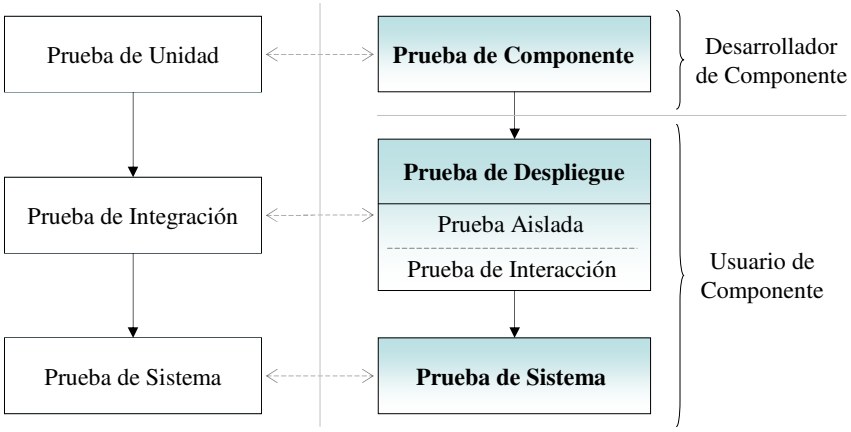


Figura 2.2: Fases de Prueba en el contexto de DSBC

forma temprana. Los componentes software se prueban inicialmente en forma aislada para detectar errores en la funcionalidad de las unidades internas al componente. La disponibilidad del código fuente para el desarrollador permite realizar pruebas de caja blanca de todas las configuraciones del componente sin considerar un contexto de uso específico [Bertolino y Polini, 2003; Jaffar-UrRehman y otros, 2007]. Además, el tamaño relativamente reducido de las unidades internas al componente permiten considerar los detalles del código cuando se determinan casos de prueba apropiados para las mismas. El desarrollador, sin embargo, debe realizar pruebas de caja negra del componente para asegurar que las especificaciones que se adjuntan al componente reutilizable, son realmente correctas. Sin embargo, tales pruebas no pueden cubrir la correspondencia funcional del comportamiento del componente a las especificaciones del sistema de destino. De hecho es imposible para el desarrollador estimar todos los posibles entornos en los cuales el componente podrá ser exitosamente insertado [Rosenblum, 1997; Weyuker, 1998; Polini y Bertolino, 2005; Gross, 2005].

Prueba de Integración El Glosario Estándar IEEE 610 [1990] define la prueba de integración como la prueba en la cual los componentes software son combinados y probados para evaluar la interacción entre ellos. De hecho la prueba de unidad no puede confirmar el comportamiento confiable de los componentes en un nuevo sistema, y requiere otro tipo de pruebas, por parte del usuario del componente durante su despliegue en un entorno real [Councill, 1999; Veenendaal, 2006].

Los componentes software se pueden incorporar a un sistema como unidades, o componentes múltiples para proveer la funcionalidad del sistema. La integración aislada o en conjunto de un componente requiere efectuar prueba de integración, es decir, probar las interacciones del componente de acuerdo a lo esperado en el entorno real de uso [Jaffar-UrRehman y otros, 2007].

Por lo tanto, la fase de prueba de integración puede también denominarse *prueba de despliegue* y podría estar conceptualmente dividida en dos sub-fases sucesivas, como se muestra en la Figura 2.2 [Polini y Bertolino, 2005; Jaffar-UrRehman y otros, 2007].

- *Prueba Aislada*

  En la primera sub-fase el componente puede ser probado como si fuera integrado en un entorno constituido de esqueletos (*drivers* y *stubs*) generalmente con mínima implementación. De esta manera se puede comprobar en forma temprana las interacciones del componente con un entorno "ideal".

- *Prueba de Interacción*

  En la segunda sub-fase, se comprueba la integración real entre varios componentes seleccionados, y para ello se pueden monitorizar las interacciones entre las implementaciones reales de los componentes arquitectónicos durante la ejecución de un TS, y posiblemente detectar interacciones indeseables. Cabe mencionar que los defectos potenciales descubiertos por el usuario del componente durante la prueba de integración no representan en general errores en la implementación. Por el contrario estos defectos pueden evidenciar la no-conformidad entre el componente esperado y el que se encuentra bajo prueba, es decir la incompatibilidad con el comportamiento esperado (y por lo tanto la necesidad de buscar otros componentes) [Gao y Wu, 2003; Polini y Bertolino, 2005].

Cuando un componente software viene equipado con un TS del desarrollador, y el usuario utiliza ese TS para re-ejecutarlo en su propio entorno, se genera un caso particular de prueba de integración. El TS provisto garantiza que las "intenciones" del desarrollador sean respetadas en el entorno final y su ejecución generalmente lleva a una evaluación más completa. Este TS puede incluir casos de prueba que no son relevantes para los propósitos específicos del usuario, aunque puede resultar útil para evaluar el comportamiento del componente bajo las entradas no-esperadas por el usuario [Polini y Bertolino, 2005; Jaffar-UrRehman y otros, 2007].

Un caso real ampliamente conocido es el fallo ocurrido en el vehículo de lanzamiento "Ariane 5" [Weyuker, 1998; Jaffar-UrRehman y otros, 2007], el cual inesperadamente cambió de dirección y explotó a menos de un minuto luego de su despegue. Este caso es una demostración desafortunada y prominente de la importancia de pruebas exhaustivas al integrar componentes software reutilizables en un sistema. En particular, el fallo se debió a la reutilización de un componente desarrollado *in-house* para la versión previa del lanzador: el Ariane 4. El componente responsable del *sistema de referencia inercial*, funcionó correctamente en todos los lanzamientos del Ariane 4, y por lo tanto no se consideró necesario realizar pruebas de integración para el caso del Ariane 5, sino que se pasó directamente a pruebas de sistema. Este caso se explica desde el punto de vista del perfil de uso definido originalmente para el componente en el Ariane 4 y el nuevo perfil de uso en el contexto del Ariane 5 [Gross, 2005].

Prueba de Sistema La prueba de sistema no presenta una gran diferencia conceptual con respecto a los procesos tradicionales y está a cargo del usuario del componente cuando todos los componentes han sido integrados y el sistema como un todo se encuentra listo para su ejecución. El proceso de la prueba de sistema no requiere pruebas de caja blanca de cada componente, aunque tiene por suposición base que cada componente ha sido probado y se ha alcanzado un cierto nivel de confiabilidad. La prueba de sistema no está restringida a probar componentes individuales, aunque los mismos son probados nuevamente en el contexto del sistema como un todo para así evaluar las cualidades esperadas del sistema, tales como rendimiento (*performance*) del sistema. Para ello se aplican estrategias de prueba de caja negra [Vincenzi y otros, 2003; Polini y Bertolino, 2005; Jaffar-UrRehman y otros, 2007].

Prueba de Regresión La prueba de regresión presenta una relación estrecha con la prueba de integración: al realizar una integración sucesiva de componentes se construye paralelamente un TS en forma incremental que detecta errores de interacción y obliga a efectuar modificaciones. Esto requiere efectuar prueba de regresión ante el nuevo contexto que se ha generado, para que en las piezas previamente probadas se compruebe nuevamente la conformidad del comportamiento de acuerdo a lo alcanzado previo a la modificación [Binder, 2000; Cai y otros, 2005]. Para ello se puede utilizar el TS generado incrementalmente, el cual debe ser extendido para la verificación de las capacidades nuevas o cambiadas de los componentes modificados [Wu y otros, 2000; Jaffar-UrRehman

y otros, 2007].

El caso citado previamente sobre el fallo del lanzador "Ariane 5" se debió en parte a la suposición que el código previamente probado debería funcionar cuando fuera reutilizado, obviando la necesidad de realizar prueba de regresión [Jaffar-UrRehman y otros, 2007].

La prueba de regresión alcanza su mayor relevancia cuando los sistemas desplegados han pasado el proceso de entrega (*release*) y por lo tanto ingresan a la etapa de mantenimiento. Un tipo común de modificación en sistemas basados en componentes se relaciona con el agregado de nuevas características, o bien el remplazo de un componente por otro debido a actualizaciones (nueva entrega) por parte de los proveedores. Aun en sistemas que no han sido concientemente diseñados para utilizar componentes de terceras partes, si un grupo de cambios se limita a un único componente, el efecto es como si un componente fuera reemplazado por otro. De ello, surge el interrogante si es posible confinar la prueba de regresión que se enfoca a nivel de componente, sólo al área del componente reemplazado o modificado; o bien, si todo el sistema debería ser probado nuevamente [Wu y otros, 2000; Weide, 2001].

Dado los costes asociados con la re-ejecución de un TS en cada escenario de modificación que sufre un sistema, generalmente se utilizan criterios para determinar cuán adecuado es el TS con respecto al objetivo de prueba, lo cual permite realizar una selección y priorización de casos de prueba. Sin embargo, en el contexto de DSBC, resulta un desafío establecer un criterio de selección adecuado debido a la usual falta de código fuente para los componentes reutilizables. Por ejemplo un enfoque tradicional basado en código no podría ser aplicable en este contexto [Rosenblum, 1997; Hou y otros, 2007].

En particular, Orso y otros [2007] han identificado enfoques de prueba de regresión que presentan estrategias de selección de casos de prueba basados en información que es obtenida de los propios componentes, mediante metadatos que los desarrolladores han instrumentado en los componentes, o bien mediante introspección o reflexión (mecanismo que permite inspeccionar la estructura e interfaces de un componente en ejecución).

2.2.3.2. Estrategias de Prueba para Componentes

Un componente puede ser clasificado dependiendo del volumen y calidad de la información que es entregada con el propio componente, y permite distinguir un espectro variado de componentes donde en un extremo encontramos el caso de componentes

completamente documentados cuyo código fuente es accesible (por ejemplo componentes *in-house* y componentes de código abierto), mientras que en el otro extremo la única información disponible para los componentes consiste de las signaturas de los servicios provistos (como en la mayoría de los ejemplos de componentes COTS)

Por lo tanto, algunas técnicas de prueba para componentes asumen la accesibilidad al código fuente, mientras que otras descartan tal disponibilidad pero dependen de que el desarrollador incorpore información de prueba o que el usuario pueda extraer cierta información de los componentes [Jaffar-UrRehman y otros, 2007].

A continuación se identifican las estrategias de prueba en el contexto de DSBC, estableciendo una relación con estrategias tradicionales que fueran directamente aplicables o con alguna adaptación en este contexto.

Prueba Funcional Dado que las estrategias funcionales no requieren de código fuente, el usuario del componente puede utilizarlas directamente, y en particular la estrategia de Partición por Categoría [Ammann y Offutt, 1994] es la más indicada al haber sido creada para el paradigma OO, que presenta claras similitudes con el paradigma basado en componentes. Sin embargo, como ocurre con todas las estrategias de caja negra, resulta necesario disponer de una adecuada especificación que provea información sobre las funciones (y argumentos) que son implementados en los servicios provistos por un componente. Por otro lado, la aplicación de estrategias de caja negra no asegura que las partes esenciales de la implementación puedan ser adecuadamente cubiertas [Binder, 2000; Vincenzi y otros, 2003].

Cuando el usuario del componente no cuenta con documentación sobre las funciones y comportamiento del componente, puede basarse en la estrategia de prueba de la interfaz (*interface probing*) [Andrews y otros, 2002] como medio intuitivo para comprender las propiedades, funcionalidad y posibles limitaciones. Esta estrategia de hecho requiere el diseño exploratorio del TS, basado en el "modelo mental" del usuario y sus expectativas sobre el componente, distinguiendo entre casos de prueba para encontrar una entrada que permita exhibir una propiedad, otros para detectar si una entrada viola una propiedad, y otros para identificar pre-condiciones del componente. Esta exploración tiene el objetivo de construir especificaciones para luego aplicar estrategias funcionales tales como Análisis de Valores Límite, entre otras [Jaffar-UrRehman y otros, 2007].

Prueba basada en Estado Si los componentes proveen especificaciones descritas en la forma de un FSM, se puede construir un FSM extendido integrando las especificaciones y luego aplicar los criterios basados en estado [Vincenzi y otros, 2003]. Pero además se pueden utilizar para construir una especie de grafo de flujo de control de la integración del comportamiento de todos los componentes que forman el sistema. El usuario del componente puede entonces aplicar una adaptación de las estrategias de prueba tradicionales de flujo de control y flujo de datos para así construir el TS [Beydeda y Gruhn, 2001; Beyeda y Ghrun, 2005].

Para ambas estrategias, el nivel de detalle de cada FSM puede tener un impacto sobre el esfuerzo de prueba. Dado que el problema de escalabilidad puede ser alto generando la intratabilidad de las especificaciones integradas [Briand y otros, 2006].

Prueba basada en Defectos En el caso de componentes que no proveen acceso al código fuente, aún se pueden aplicar ciertas estrategias basadas en defectos, particularmente en función de las interfaces provistas por los componentes. Así la técnica de mutación de interfaz [Delamaro y otros, 2001; Gosh y Mathur, 2001] se puede utilizar con mínimos ajustes en el contexto de DSBC [Vincenzi y otros, 2003].

Por ejemplo, Hou y otros [2007] proponen una aplicación de mutación de interfaz para prueba de regresión considerando *diseño basado en contratos*, dado que los contratos definidos en las interfaces de los componentes reutilizables pueden describir pre- y post-condiciones de los servicios de los componentes y actuar de "intermediarios" entre el proveedor del componente y el usuario del componente.

Por otro lado, Yoon y Choi [2004] proponen utilizar los operadores mutantes de interfaz para la técnica de inyección de errores, los cuales están enfocados en las interacciones con el componente a través de su interfaz. Para ello se aplica un proceso similar a la fase de prueba de integración (*prueba de despliegue*), descrita en la Sección 2.2.3.1 (pág. 48), analizando el componente primero en forma aislada y luego en presencia de arquitecturas reales y la integración a otros componentes.

Prueba basada en Metadatos Una alternativa para las pruebas se basa en el concepto de metadatos, que consiste de varias formas de datos estáticos, tales como datos de versionado, dependencias de datos y control, representaciones abstractas de código fuente, o métricas de complejidad; y algunos datos dinámicos tales como cubrimientos

estructurales logrados cuando se ejecutan los servicios del componente. Estos metadatos pueden ser calculados o recuperados mediante servicios especializados denominados meta-métodos.

Algunos frameworks de componentes actuales, incluyendo Enterprise JavaBeans [DeMichil y otros, 2001], Java 2 Enterterprise Edition [J2EE, 2008], la tecnología COM [COM, 2008] (DCOM,COM+), y la iniciativa .NET [Wigley y otros, 2003; .NET, 2009] ya proveen alguna información a través de datos empaquetados con los componentes y recuperables a través del mecanismo de reflexión.

El enfoque basado en metadatos tiene el objetivo de mejorar la facilidad de prueba de los componentes. Sin embargo, estos enfoques requieren: 1) que el vendedor del componente esté involucrado en su producción, 2) que esta información sea empaquetada con el componente en una forma estándar, y 3) que pueda ser procesada por herramientas automáticas.

Dependiendo de lo anterior, en el caso de componentes de mayor envergadura puede resultar demasiado compleja su aplicación [Vincenzi y otros, 2003; Briand y otros, 2006; Orso y otros, 2007; Jaffar-UrRehman y otros, 2007].

En particular, Orso y otros [2007] han identificado estrategias basadas en metadatos enfocadas en prueba de regresión, clasificándolas de acuerdo al tipo de información que utilizan: basadas en código (a nivel de sentencia, método y componente), y basadas en especificaciones descritas con FSMs.

Auto-Prueba de Componentes Generalmente el desarrollador de componentes construye un componente considerando un perfil específico de uso, que es sometido a pruebas con diferentes configuraciones. Sin embargo no se puede anticipar todo posible contexto real de uso que puede afectar algunas de las capacidades diseñadas para el componente. Como ejemplo se puede mencionar nuevamente el caso del fallo en el Ariane 5, donde un componente reutilizado del Ariane 4 fue integrado sin efectuar pruebas ante el nuevo entorno de ejecución [Gross, 2005; Jaffar-UrRehman y otros, 2007].

Un enfoque alternativo es dotar a los componentes con capacidades adicionales de verificación del entorno al cual será integrado. Esta estrategia se conoce como BIT (*built-in-testing*), y el objetivo es que un componente pueda por sí mismo comprobar si se comporta en la forma esperada cuando es insertado en un sistema. En esencia para cada componente se provee una interfaz sencilla que permite encapsular o "decorar" el

componente con capacidades BIT sofisticadas. Tales capacidades adicionales que provee el componente decorador (*wrapper*) se relacionan a responsabilidades de auto-comprobación y auto-prueba las cuales verifican la conformidad de contratos de los componentes servidores (a los cuales debe solicitar colaboración, y que correspondería a las interfaces requeridas), incluyendo la plataforma subyacente [Edwards, 2001; Jaffar-UrRehman y otros, 2007].

Esta estrategia puede incluir algunas de las siguientes características: 1) los wrappers BIT son completamente transparentes al código del cliente y del propio componente, y pueden ser insertados o retirados sin producir cambios en el código cliente, y sin generar un costo en ejecución para el propio componente; 2) se pueden verificar aserciones internas y externas sobre el comportamiento del componente, que pueden ser especificadas con un modelo abstracto del componente, en lugar de hacerlo en términos de las estructuras de representación interna; 3) se pueden detectar ciertas violaciones en el momento en que ocurren y antes que puedan propagarse a otros componentes, identificando el origen específico de la violación (es decir el método/operación responsable); 4) se puede lograr una observación completa del estado interno del componente sin romper el encapsulamiento para los clientes.

Esta estrategia requiere por parte de los desarrolladores un esfuerzo adicional para instrumentar los componentes con tales capacidades de auto-verificación basadas en aserciones, y/o un TS específico. Por ejemplo el enfoque de Edwards [2001] está basado en una especificación formal en el lenguaje RESOLVE, mientras que el enfoque de Atkinson y otros [2003] está basado en una especificación en KobrA. Estas especificaciones son utilizadas para luego construir aserciones que serán instrumentadas en los componentes para la verificación del componente mientras se realiza la ejecución del TS, también provisto con el componente.

Capítulo 3

Proceso de Evaluación de Componentes Software

3.1. Introducción

Los usuarios generalmente asumen una garantía en la disponibilidad de la funcionalidad que utilizan diariamente, lo cual genera un desafío para los ingenieros de software en el sentido de mantener apropiadamente la estabilidad esperada ante las continuas actualizaciones que puede requerir un sistema. Por ello, cualquier componente de reemplazo –es decir, una nueva versión entregada (*release*) o un nuevo componente adquirido– debe ser cuidadosamente manejado, aun cuando sea provisto por el mismo vendedor. La principal inquietud procede generalmente de la posible pérdida de alguna funcionalidad, aunque también es posible que la innovación en algunas funciones pueda acarrear ciertos efectos colaterales al sistema.

En consecuencia, la preocupación principal de un ingeniero de software es contar con un mecanismo para efectivamente seleccionar un determinado componente de un conjunto de componentes candidatos, e identificar si el componente de reemplazo seleccionado puede sustituir a otro actualmente integrado en un sistema basado en componentes. Para llevar a cabo esto, no resulta suficiente solamente con efectuar un análisis de equivalencia de interfaces de los componentes, sino que además se debe realizar una inspección profunda de su comportamiento para distinguir probables efectos imperceptibles que pudieran incluso producir la pérdida de alguna otra funcionalidad presente en el sistema.

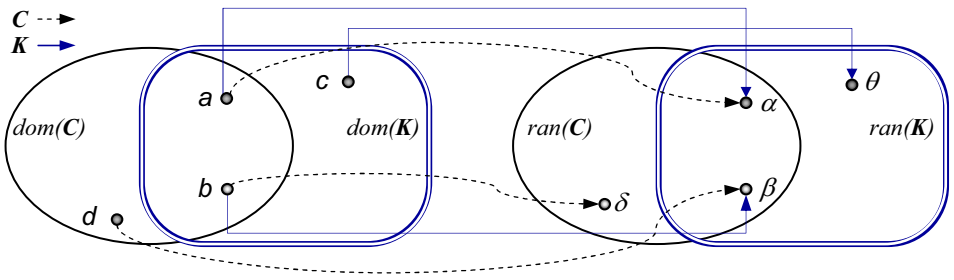


Figura 3.1: Mapeo Funcional de los componentes *C* y *K*

En particular para observar el comportamiento de los componentes se analizan sus funciones internas de transformación de datos, según la métrica de pruebas *facilidad de observación*, que como se explica en la Sección 2.2.2.1 del capitulo anterior, se enfoca en el comportamiento operacional de un componente distinguiendo su salida como función de su entrada [Freedman, 1991; Jaffar-UrRehman y otros, 2007]. Cuando se expone tal percepción de comportamiento de dos componentes diferentes, y luego se extrae la información adecuada para un procedimiento de comparación, es posible identificar el grado de compatibilidad existente entre ambos, como lo han discutido Alexander y Blackburn [1999], y Cechich y Piattini [2007].

Dado un componente *C* y un componente de reemplazo candidato *K*, cada uno de ellos acepta un cierto conjunto de datos de entrada (dominio) a través de sus servicios, desde donde una función de transformación interna retorna un conjunto de datos de salida (rango). Si ambos componentes fueran equivalentes, es esperable que tanto el dominio como el rango en ambos componentes también fuera equivalente. Sin embargo, existe una inquietud adicional relacionada con la exacta generación de un dato de salida particular desde un dato de entrada específico, lo cual se conoce como *mapeo funcional* entre el dominio y el rango. Por lo tanto, es aún más importante encontrar equivalencia en el mapeo funcional para ambos componentes. En la Figura 3.1 se muestra un caso donde hay intersección no vacía en el dominio y el rango, aunque no coinciden completamente, lo cual tampoco ocurre con las vinculaciones en el mapeo funcional. Por ejemplo, para el dato *b* de la intersección de dominios, las salidas correspondientes no coinciden – β para *K* y δ para *C*, lo cual se puede explicar como que *K* está '*semánticamente distante*' de *C*. Un caso como éste debería ser identificado apropiadamente para evitar una situación no deseada y así lograr mantener la estabilidad esperada de un sistema.

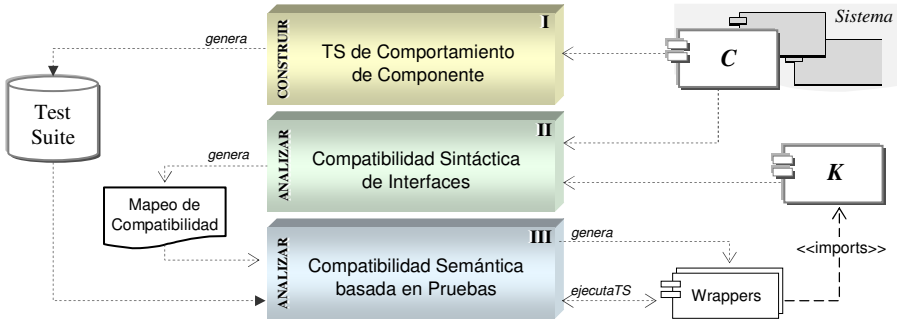


Figura 3.2: Proceso de Evaluación para la Sustitución de Componentes Software

Mientras que realizar una exploración completa del mapeo funcional puede ser demasiado extenso, enfocarse en aspectos específicos y datos representativos resulta más eficiente mientras que continúa siendo adecuadamente efectivo. Para lograr esto, se puede realizar una selección apropiada de criterios de cobertura de prueba, lo cual resulta en una forma ágil de sustentar el análisis de mapeo funcional, y que ha sido la opción que se ha seguido en este enfoque.

Nuestra propuesta para evaluación de componentes software está conceptualmente basada en la técnica de Pruebas Back-to-Back, en la cual dos o mas variantes de un componente o sistema (es decir, que se los supone funcionalmente equivalentes), son ejecutados con las mismas entradas, comparando las salidas, y analizado si existe correlación entre las probabilidades de falla [Vouk y otros, 1987; Shimeall y Leveson, 1991; Veenendaal, 2006]. En el caso que se necesita probar una nueva implementación de un componente, se puede juzgar su correctitud en función de una implementación anterior existente que sirve de referencia. Se genera un conjunto de casos de prueba (TS, de *test suite*) para la implementación de referencia, y luego se ejecuta ese TS contra ambas implementaciones para comparar los resultados [Edwards, 2001].

En esencia, el Proceso de Evaluación para la Sustitución de Componentes Software, que proponemos en esta tesis, ha sido definido en función de tres fases principales que se representan en el modelo contextual de la Figura 3.2 y luego se detallan en el diagrama de actividad de la Figura 3.3. Siendo C un componente original y K un posible componente de reemplazo, el proceso completo involucra lo que se describe a continuación:

1^{ra} Fase. Se genera un TS con el propósito de representar aspectos de comportamiento de un componente C que requiere ser reemplazado. Luego se debe comprobar si

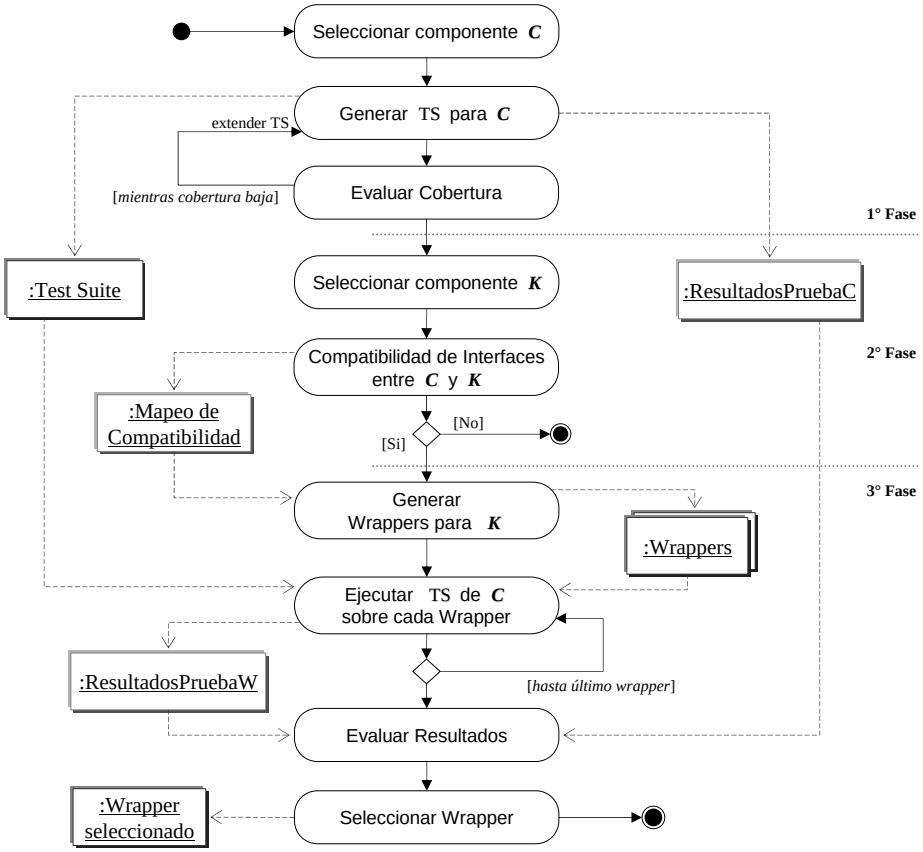


Figura 3.3: Diagrama de actividad del Proceso de Evaluación

el TS es adecuado, para lo cual se mide la cobertura alcanzada por los casos de prueba contra el componente *C*, aplicando los criterios de cobertura indicados en la Sección 2.2.2.2 (pág. 46). Es importante notar que el objetivo de este TS no es encontrar fallos, sino representar comportamiento y, en particular, describir las probables interacciones del componente *C* con otros componentes en el sistema software. Esto será convenientemente explicado en la Sección 3.2 (pág. 62).

2^{da} Fase. Se comparan a nivel sintáctico, las interfaces ofrecidas por el compomente *C* y por el componente candidato *K*. Si el componente *K* ofrece una interfaz compatible con aquella del componente *C*, entonces *K* puede pasar a la siguiente fase. El análisis realizado considera si el conjunto de servicios ofrecidos por *K* contienen los servicios ofrecidos por el componente *C*. A este nivel, puede existir compatibilidad aún cuando los servicios de *C* y *K* tengan diferentes nombres, diferente orden en la lista de parámetros, etc. El producto resultante de esta fase es un Mapeo de Com-

patibilidad donde cada servicio del componente *C* puede tener correspondencia con uno o más servicios del componente *K*. Los detalles de esta fase son ampliados en la Sección 3.3 (pág. 73).

3ra Fase. El componente *K* que ha pasado el análisis de compatibilidad de interfaces debe ser evaluado a nivel semántico, lo cual implica ejecutar el TS generado para el componente *C* en la primera fase, contra el componente *K*. El propósito es encontrar las verdaderas correspondencias de servicios del Mapeo de Compatibilidad generado en la segunda fase, y por ello se lo procesa para generar un conjunto de wrappers W (adaptadores) para *K*. El objetivo final es encontrar un wrapper $w \in W$ que pueda reemplazar a *C* para permitir que los actuales clientes de *C* puedan interactuar en forma segura con la interfaz de *K*. Para lograr esto, cada wrapper $w \in W$ se va tomando por vez como la clase bajo prueba y se lo ejercita con el TS que se construyó para *C* en la primera fase. Una vez que todo el conjunto W ha sido probado, se analizan los resultados obtenidos para revelar si se puede concluir la existencia de una compatibilidad a nivel semántico. Si esto sucede, implicará que al menos un wrapper $w \in W$ pueda ser seleccionado como el más adecuado para permitir el ajuste o adaptación de *K*, para que pueda ser integrado al sistema como un reemplazo de *C*. Detalles de esta fase se explican en la Sección 3.4 (pág. 85).

Las próximas secciones proveen información detallada de cada fase. La aplicación del proceso se ilustrará mediante el siguiente caso de estudio.

3.1.1. Ejemplo

Para ilustrar el proceso tomaremos un ejemplo sencillo y ampliamente conocido sobre un sistema bancario (Figura 3.4) adaptado de Orso y otros [2000], donde uno de los componentes que maneja las cuentas de los clientes, representado en este caso por la clase `Account`, debe ser reemplazado. Se puede observar la interfaz del componente en la Figura 3.5(a). Se cuenta con un componente candidato para efectuar el reemplazo, que podría considerarse una versión en castellano del componente original, cuya clase principal se denomina `Cuenta`, y la interfaz del componente se muestra en la Figura 3.5(b).

Con el propósito de lograr una decisión conclusiva sobre la compatibilidad entre `Account` and `Cuenta`, se debe iniciar la primera fase del proceso de evaluación. Como se ha explicado previamente, esto involucra la creación de un TS de Comportamiento de

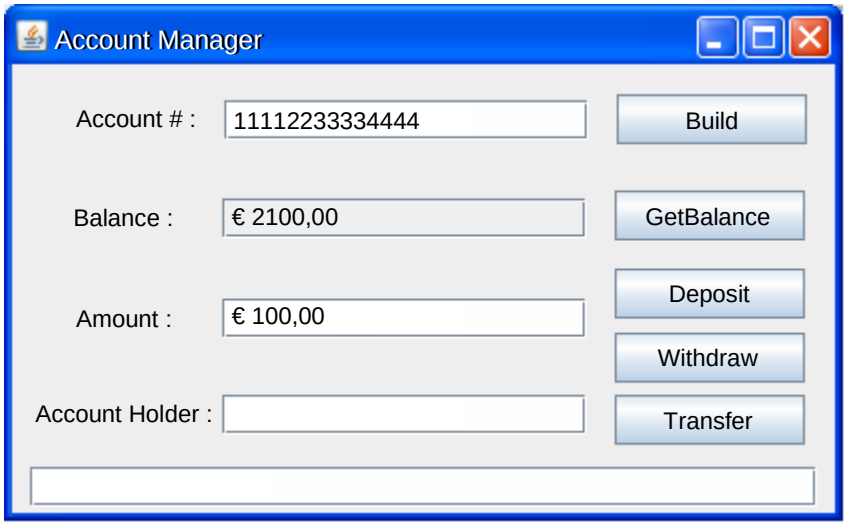
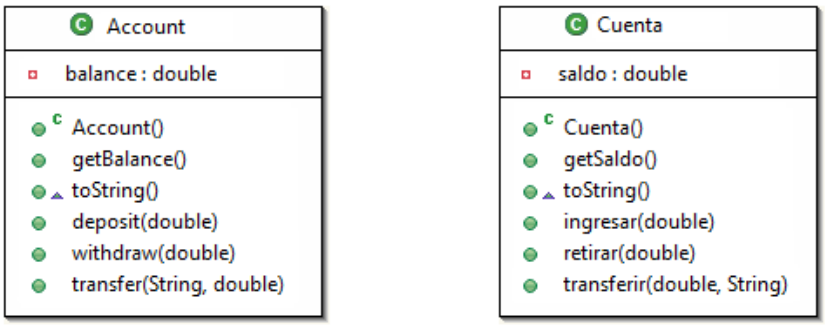


Figura 3.4: Ejemplo *Sistema Bancario* en Java



(a) Componente Original (b) Componente de Reemplazo

Figura 3.5: Componente original `Account` (a) para ser sustituido por `Cuenta` (b)

Componente para `Account`, el cual será más tarde utilizado para la evaluación semántica. Volveremos a este ejemplo en las siguientes secciones, con el fin de ilustrar las fases del proceso de evaluación.

3.2. TS de Comportamiento de Componente

Para construir un TS como una representación de un componente, se ha seleccionado un conjunto de criterios de cobertura específicos para pruebas de componentes. El objetivo de este TS es comprobar que un componente candidato K coincide en

comportamiento con un componente original dado C. Por lo tanto, cada caso de prueba en el TS consistirá de un conjunto de invocaciones a servicios en la interfaz de C, desde donde los resultados de las pruebas son almacenados en un repositorio para determinar la aceptación o rechazo cuando el TS es aplicado contra el componente K.

En base a los conceptos sobre criterios de cobertura de pruebas para componentes introducidos en la Sección 2.2.2.2 (pág. 46), a continuación se explica la estrategia seleccionada para su implementación en nuestro enfoque. Esto está principalmente relacionado con los "*eventos*" que se producen en los componentes y los efectos resultantes (es decir, la invocación a una interfaz), considerando que los eventos pueden ser síncronos tales como las invocaciones a los servicios, o asíncronos tales como las excepciones.

El TS de Comportamiento de Componente ha sido encuadrado en la categorización de *dependencia intra-componente*, es decir que los eventos (servicios o excepciones) de la interfaz del componente presentan dependencia con otros eventos de la misma interfaz, y por lo tanto el comportamiento de cada evento puede variar dependiendo de cuáles eventos (de la misma interfaz) ocurrieron previamente. Esto implica que inicialmente sólo se necesita describir la inter-dependencia de los eventos dentro de la propia interfaz de un componente, y que no estarían involucradas interfaces externas, por lo que no se requiere la presencia de otro componente durante las pruebas. Así, el enfoque resulta de menor complejidad para la evaluación de componentes sin la necesidad de requisitos adicionales de otra plataforma o entorno.

Además de estar categorizado con *dependencia intra-componente*, el TS de Comportamiento de Componente ha sido también modelado para cubrir el *criterio de dependencia del contexto*, el cual ayuda a describir las interdependencias dentro de la interfaz de un componente por medio de secuencias de eventos (servicios y excepciones). Tales secuencias de eventos (o secuencias operacionales) en nuestro enfoque son formalizadas por medio de *expresiones regulares*, dado que ello nos permite la automatización de la fase de generación de casos de prueba. El alfabeto requerido por las expresiones regulares se compone de las signaturas de los servicios de los componentes.

Dado que las expresiones regulares permiten formalizar un patrón general para las secuencias operacionales, el resultado se puede denominar "*protocolo de uso*" para la interfaz de un componente –similar al concepto definido por Kirani [1994] en el contexto de las pruebas de clases para orientación a objetos. Esta formalización de las secuencias operacionales como la manera de lograr una implementación válida para el *criterio de de-*

pendencia del contexto para componentes posee una justificación concreta que se describe a continuación.

3.2.1. Criterios base para el TS de Comportamiento

Mariani y otros [2004] han propuesto criterios de cobertura específicos para expresiones regulares teniendo en cuenta sus elementos constitutivos, tales como el alfabeto, sus operadores, y la posibilidad de derivar subexpresiones. La Tabla 3.1 presenta los criterios de cobertura para expresiones regulares, de los cuales hemos ampliado el *criterio de operadores* de acuerdo a lo expuesto por Zakhour y otros [2006] dado que Mariani y otros [2004] sólo propusieron el uso del operador Kleen '$*$' (entre los cuantificadores u operadores de iteración), y al posibilitar la utilización de otros cuantificadores se pueden especificar *protocolos de uso* con mayor detalle, lo cual a su vez permite mejorar la descripción de comportamiento de los componentes. Los criterios para expresiones regulares exhiben una relación con los criterios de cobertura para componentes presentados en la Sección 2.2.2.2 (pág. 46), y permiten justificar por qué el uso de expresiones regulares resulta una estrategia adecuada en este enfoque. A continuación se explica esta relación entre tales criterios, la cual además puede observarse en la Figura 3.6.

Dado que los servicios de un componente proveen los símbolos para el alfabeto de las expresiones regulares para lograr la formalización de las secuencias operacionales, el *criterio de alfabeto* permitiría invocar todos los servicios de un componente, lo cual cubre por tanto el *criterio de métodos*. Por otro lado, el conjunto de operadores para expresiones regulares (tales como '|', '$*$', etc), ayudan a describir patrones generales que incluyen cada uno de los casos de secuencias operacionales, por lo que el *criterio de operadores* podría considerarse casi equivalente al *criterio de dependencia del contexto*. Sin embargo, como únicamente se han considerado los servicios de un componente en el *protocolo de uso*, todavía no se alcanza a cubrir completamente el *criterio de eventos* en el cual se deben considerar también las excepciones (eventos asíncronos).

En nuestro enfoque, las excepciones inicialmente no se especifican en el *protocolo de uso*, sino que se describen posteriormente por medio de su vinculación a la invocación de un servicio, en conjunto con los datos particulares para los cuales ocurrirá una excepción. Así cada excepción finalmente queda inserta en una secuencia operacional, la cual incluye el servicio al que se vincula la excepción. Y de esta manera se cubre el *criterio de ex-*

Criterios para expresiones regulares	**Ejemplo:** $a^*b(b\|c)$
Alfabeto: Para cada símbolo en el alfabeto debe haber un caso de prueba que lo contenga	TS= $\{abc\}$ satisface el criterio de *alfabeto* para la expresión regular del ejemplo
Operadores: Para cada operador: ▪ '|' (alternativa) debe existir al menos un caso de prueba que contenga el primer operando y algún otro que contenga el segundo operando ▪ '+' debe existir al menos un caso de prueba que corresponda a exactamente una iteración y al menos otro para más de una iteración del operando ▪ '$*$' (kleen) similar al anterior y además debe existir al menos un caso de prueba que corresponda a ninguna iteración del operando ▪ '?' debe existir al menos un caso de prueba que corresponda a exactamente una iteración y al menos otro que corresponda a ninguna iteración del operando	TS= $\{bb, abc, aabb\}$ satisface el criterio de *operadores* para la expresión regular del ejemplo
Expresiones: Para cada elección de operadores que resulta en diferentes sentencias quizá con longitudes distintas, debe existir al menos un caso de prueba que las cubra	TS= $\{bb, bc, abb, abc, aabb, aabc\}$ satisface el criterio de *expresiones* para la expresión regular del ejemplo

Tabla 3.1: Criterios de Cobertura para Expresiones Regulares

cepciones, y por tanto el *criterio de eventos*, lo cual permite que el *criterio de operadores* provea una cobertura equivalente al *criterio de dependencia del contexto* para componentes (como se observa en la Figura 3.6).

El *criterio de expresiones* para expresiones regulares implica cubrir toda posible sentencia derivable, lo cual considerando la invocación a servicios y excepciones, implicaría que toda posible combinación de comportamientos de un componente estaría cubierta por este criterio. Sin embargo, esto resulta en general en un TS de una magnitud demasiado grande, que imposibilita su manejo y quizá también su generación. Además de ello, una gran mayoría de los casos de prueba de tal TS corresponderían a configuraciones equivalentes, lo cual no suministra información significativa. Por esta razón, el *criterio de expresiones* no se considera en nuestro enfoque, lo cual sin embargo no perjudica la calidad del TS de Comportamiento de Componente, que está basado principalmente en la representación de aspectos significativos.

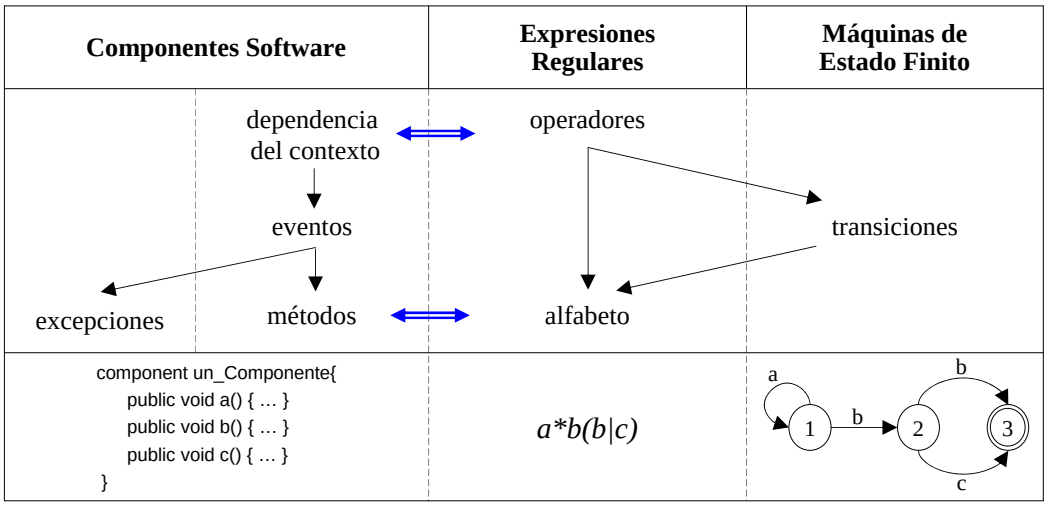


Figura 3.6: Relación de subsumción y equivalencia entre distintos criterios

Las secuencias operacionales pueden también ser derivadas a partir de Máquinas de Estado Finito, las cuales son de hecho ampliamente usadas en el campo de las pruebas del software para representar comportamiento y derivar casos de prueba [Binder, 2000; UML 2.0, 2003; Kirani y Tsai, 1994]. En el contexto de las secuencias operacionales, las signaturas de los servicios podrían ser representados como arcos en una máquina de estados, como se puede observar en la Figura 3.6 donde se ejemplifica con la expresión regular de la Tabla 3.1. Por supuesto las máquinas de estados también pueden ser usadas para representaciones más complejas del comportamiento de un componente describiendo los estados abstractos y la forma como estos son alcanzados y atravesados.

Dado que las máquinas de estado pueden ser a su vez representadas por expresiones regulares, se puede identificar una equivalencia o relación de subsumción entre los criterios de cobertura para ambas notaciones. Por ejemplo el *criterio de transiciones* para máquinas de estado (denominado *criterio de arcos* por Mariani y otros [2004]) subsume al *criterio de alfabeto*. Sin embargo el *criterio de operadores* ha sido definido con una cobertura mucho mas amplia y por lo tanto subsume al *criterio de transiciones*, como se muestra en la Figura 3.6.

3.2.2. Metamodelo para Generar el TS

Con respecto a la implementación concreta para diseñar el TS de Comportamiento de Componente y representar adecuadamente los criterios de cobertura, se ha optado por considerar modelos de componentes que suministren algún mecanismo de introspección. Entre ellos los frameworks de Java y la tecnología .Net, poseen el mecanismo de reflexión que permite extraer los elementos de la interfaz de un componente.

De esta manera, las signaturas de los servicios extraídas de la interfaz de un componente constituyen los símbolos del alfabeto para las expresiones regulares, mientras que las excepciones recolectadas de los servicios (también por medio de reflexión), permiten fortalecer la representación de comportamiento de los componentes. De hecho, algunas excepciones requieren que un componente se encuentre en un estado específico al que sólo se arriba luego de previas ejecuciones de otros eventos (tales como invocaciones a ciertos servicios), por lo tanto las secuencias operacionales (*dependencia del contexto*) son generalmente la única estrategia plausible para obtener una apropiada cobertura.

Para la generación de casos de prueba se construyó un metamodelo que permite representar y administrar los elementos constitutivos de un TS. La Figura 3.7 presenta una vista parcial del metamodelo, en el cual se muestra la clase *TClass* que representa la clase bajo prueba (CUT, de *class under test*), la cual tiene asociados sus campos (*TField*) y sus operaciones (*TOperation*). Se puede observar que entre las operaciones, la clase *TMethod* (que representa la información extraída de la interfaz del CUT), tiene acceso a la clase *Method* que pertenece al metamodelo base de orientación a objetos y que justamente permite (por medio de reflexión) acceder a los datos de sus instancias.

Los casos de prueba se generan en dos formatos: JUnit [JUnit Home Page, 2008] y MuJava [μJava Home Page, 2008], dado que cada uno de ellos permite concebir diferentes propósitos: los casos de prueba JUnit permiten encontrar fallos aplicando técnicas de prueba de caja negra, mientras que aquellos en formato MuJava son de caja blanca y permiten (entre otros propósitos) medir la cobertura de un TS mediante el criterio de mutantes muertos [Polo y otros, 2008].

Inicialmente se genera el TS en formato JUnit, es decir una instancia de la clase *JUnitFile* que es una especialización de *TestFile* (en el metamodelo de la Figura 3.7). El propósito de este TS en formato JUnit, es validar convenientemente el TS por medio de la ejecución de sus casos de prueba contra el componente original C. Esta

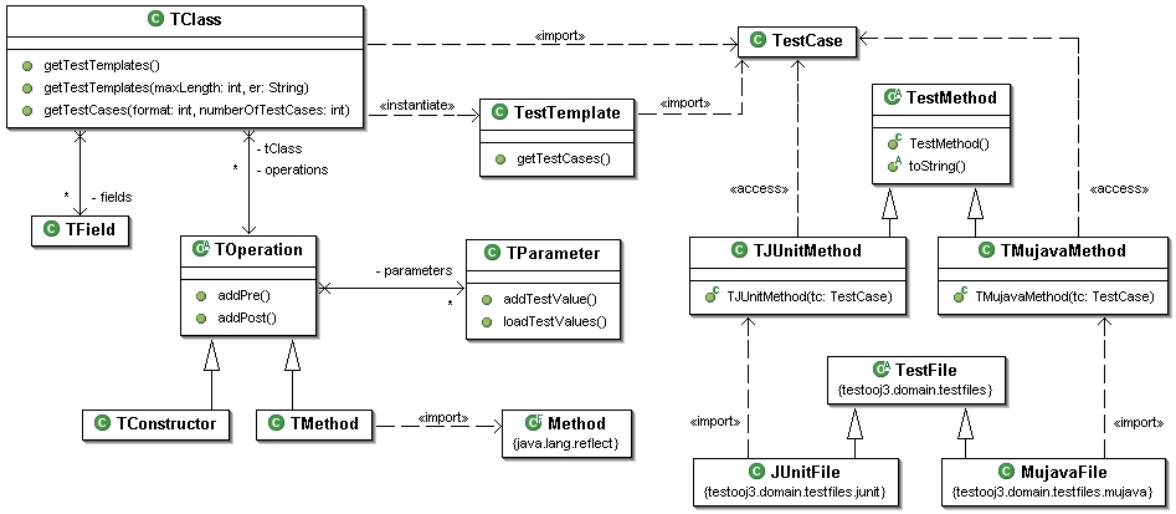


Figura 3.7: Vista parcial del Meta-modelo para generación de Casos de Prueba

validación se alcanza cuando se obtiene un 100 % de resultados exitosos, dado que el TS está diseñado para incluir configuraciones de casos de prueba que no fallan o bien que provocan excepciones controladas. De esta manera, el TS alcanza el objetivo de representar diferentes facetas de comportamiento del componente original C.

Luego que el TS ha sido apropiadamente validado, se deriva un TS equivalente en formato MuJava, es decir una instancia de la clase *MuJavaFile* (en el metamodelo de la Figura 3.7), para que pueda ser usado en la tercera fase del proceso de compatibilidad, dado que esto facilita el análisis que allí se realiza (esto será explicado con mayor detalle en la Sección 3.4, pág. 85).

Por medio del ejemplo presentado en la sección previa, a continuación se explica cómo se lleva a cabo el procedimiento de construcción del TS de Comportamiento de Componente, analizando cómo se alcanza la cobertura de los criterios correspondientes.

3.2.3. TS para Account

Para construir un TS de Comportamiento de Componente, se definió una serie de pasos que se pueden observar en la Figura 3.8, los cuales han sido implementados por el doctorando en la herramienta *testooj* [Polo y otros, 2007], que permite en particular tratar con componentes Java, e incluye el metamodelo descripto en la sección previa

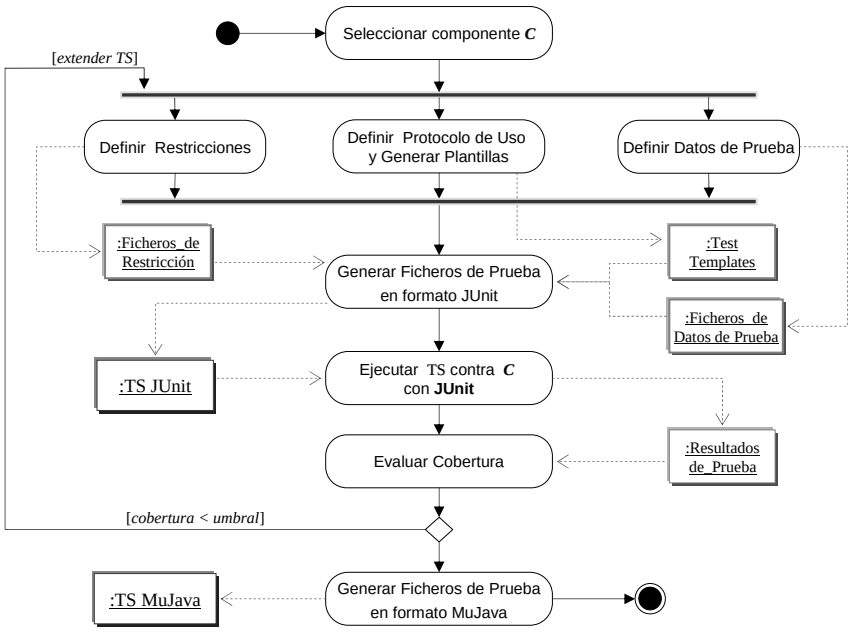


Figura 3.8: Generación del TS de Comportamiento de Componente

como una capa subyacente. Así como parte del desarrollo de esta tesis, también se han automatizado los pasos que comprende esta fase, que además de facilitar la validación del enfoque, permiten principalmente alcanzar el objetivo de suministrar un proceso más riguroso.

Uno de los pasos inciales para construir el TS para `Account`, de acuerdo a la Figura 3.8, implica definir el *protocolo de uso* (en la forma de una expresión regular), que para `Account` podría asumir el siguiente formato:

$$\texttt{Account (getBalance | deposit | toString)}$$
$$\texttt{(getBalance | deposit | toString | withdraw | transfer)}^*$$

De la expresión regular anterior, se derivan sentencias que permitan crear plantillas de prueba (*test templates*), las cuales describen las diferentes secuencias operacionales. Para generar estas plantillas se hace uso de la clase *java.util.regex.Pattern* del framework de Java, que facilita identificar si las sentencias derivadas son válidas con respecto a la expresión regular correspondiente.

Estas plantillas se crean como instancias de la clase *TestTemplate* del metamodelo de la Figura 3.7 (en la sección previa, pág. 68), y para su generación se considera qué

longitud deben tener las sentencias que se derivan de una expresión regular (cantidad de símbolos del alfabeto incluídos en una sentencia). Esto se podrá comprender mediante el siguiente análisis que se realiza en función del ejemplo, que además facilitará considerar si es posible generar las secuencias operacionales necesarias, que permitan que el TS posea la cobertura del *criterio de dependencia del contexto*.

Observando la expresión regular que formaliza el protocolo de uso para `Account`, se podría señalar que como mínimo se debería tener una longitud de 3 símbolos, lo cual deriva 18 sentencias que incluyen como máximo una sola iteración del operador Kleen (∗), y esto permite cubrir el *criterio de alfabeto* porque todos los servicios de la interfaz de `Account` se cubrirían con algún caso de prueba. A continuación se muestran algunos ejemplos de sentencias derivadas de la expresión regular:

`Account`	`Account`	`Account`	`Account`	`Account`
`getBalance`	`toString`	`deposit`	`toString`	`getBalance`
	`deposit`	`withdraw`	`transfer`	`toString`

Se puede observar que se ha cubierto el primer operador alternativa (|), y luego cómo la primer columna ejemplifica para el operador Kleen (∗) que haya cero iteración, y luego en las siguientes columnas que haya una iteración, cumpliendo además con el segundo operador alternativa (|). Sin embargo el operador Kleen (∗) requiere no solamente cero y una iteración, sino además una iteración extra (como se explica en la Tabla 3.1 de la sección previa, pág. 65), por lo tanto esa longitud mínima de 3 símbolos, no permite cubrir el *criterio de operadores*. Por ello se extiende la longitud miníma a 4 símbolos, con lo cual se generan 75 sentencias más, que incluyen una iteración adicional para el operador Kleen (∗). Así, se generaron finalmente 93 plantillas de prueba, algunas de las cuales se pueden observar en la Tabla 3.2, como por ejemplo la plantilla `testTS_75`, que luego de crear la cuenta, obtiene un reporte textual del saldo, realiza un ingreso y luego un retiro de fondos.

`testTS_1`	`testTS_57`	`testTS_75`	`testTS_86`
Account()	Account()	Account()	Account()
getBalance()	deposit(double)	toString()	toString()
deposit(double)	transfer(String,double)	deposit(double)	withdraw(double)
	deposit(double)	withdraw(double)	transfer(String, double)

Tabla 3.2: Plantillas de Prueba para Account

Método	Datos de Prueba	Restricciones y Oráculo
deposit	double (50, 100)	pre: double value = obtained.getBalance(); post: assertTrue(obtained.getBalance() == value + arg1);
withdraw	double (20, 30)	pre: double value = obtained.getBalance(); post: assertTrue(obtained.getBalance() == value - arg1);
transfer	String (andrew)	pre: double value = obtained.getBalance(); post: assertTrue(obtained.getBalance() == value - arg1);
	double (10)	

Tabla 3.3: Datos de Prueba y Restricciones para Account

Los siguientes pasos involucran establecer algunos otros aspectos como: restricciones, excepciones y datos de prueba. Para realizar la carga de los datos de prueba se debería realizar un análisis previo con el propósito de seleccionar un conjunto representativo de valores, para lo cual existen algunas técnicas útiles como Partición en Clases de Equivalencia y Análisis de Valores Límites, entre otras [Binder, 2000; Myers, 2004]. En la Tabla 3.3 se muestran a modo de ejemplo algunos datos de prueba para los servicios de `Account`, que serán luego usados y combinados de acuerdo a las plantillas de prueba.

Dado que los casos de prueba en formato JUnit no retornan un valor que permita reconocer éxito o fallo, se ha utilizado una sentencia especial que se basa en los operadores de la clase *Assert* (que provee el framework de JUnit), y de esta manera se puede comprobar el estado del CUT luego de la ejecución del caso de prueba. Estas sentencias especiales actúan como un oráculo, indicando cuándo se produce un fallo. Así en la Tabla 3.3 también se muestra el oráculo para los servicios, en los cuales se utilizó la operación *assertTrue*. El oráculo en conjunto con algún otro código auxiliar que se necesite agregar a los casos de prueba, forman las restricciones que se necesitan aplicar, y que se guardan en ficheros de restricciones (Figura 3.8, pág. 69).

Luego de estos pasos, los datos de prueba son usados en combinaciones con las 93 plantillas de prueba (secuencias operacionales) y los ficheros de restricciones. Tales combinaciones se realizan de acuerdo a uno de los cuatro algoritmos que han sido implementados en la herramienta *testooj*, que resultan particularmente útiles cuando más de un dato de prueba ha sido cargado. Los algoritmos son: *each choice* [Ammann y Offutt, 1994], *antirandom* [Malaiya, 1995], *pairwise* [Czerwonka, 2006], y *all combinations* [Grindal y otros, 2005]. En el caso práctico fue aplicado para el TS de `Account` el algoritmo *all combinations* para intentar abarcar un rango mayor de combinaciones y así

Plantilla `testTS_1`	Plantilla `testTS_75`

public void testTS_1_1() {
bank.Account obtained=null;
obtained =new bank.Account();
double result0=obtained.getBalance();
double arg1=(double) 50;
obtained.deposit(arg1);
double value = obtained.getBalance();
assertTrue(obtained.getBalance() == value + arg1);
}
	public void testTS_75_1() {
	bank.Account obtained=null;
	obtained =new bank.Account();
	java.lang.String result0=obtained.toString();
	double arg1=(double) 50;
	obtained.deposit(arg1);
	double value = obtained.getBalance();
	assertTrue(obtained.getBalance() == value + arg1);
	double arg2=(double) 20;
	obtained.withdraw(arg2);
	double value = obtained.getBalance();
	assertTrue(obtained.getBalance() == value - arg2);
	}

Tabla 3.4: Casos de Prueba en formato JUnit para Account

asegurar que se cubren más areas de prueba, lo cual implica (en el ejemplo) cubrir más casos de interacción de los potenciales clientes con el componente `Account`.

Cada combinación se convierte en un caso de prueba, que asume la forma de un método (instancia de la clase *TJUnitMethod* en el metamodelo de la Figura 3.7, pág. 68) dentro de un fichero manejador de pruebas, el cual es serializado y almacenado en un repositorio. En el caso de `Account`, se generaron 228 casos de prueba como métodos dentro de un clase denominada `JUnitAccount` (instancia de la clase *TJUnitFile* en el metamodelo). La Tabla 3.4 muestra ejemplos de casos de prueba en formato JUnit para las plantillas `testTS_1` y `testTS_75` que se presentan en la Tabla 3.2. Se puede observar cómo se han combinado las plantillas con datos de prueba y restricciones.

El paso siguiente luego que el TS ha sido generado es validarlo mediante su ejecución contra el componente `Account` (para el cual fue originalmente construido). Para ello se utiliza la herramienta de ejecución de JUnit con la clase `JUnitAccount` (que representa el TS generado) e iterando a través de sus casos de prueba. La evaluación de los casos de prueba se realiza de acuerdo a la operación *Assert* que los mismos contienen y que actúa como el oráculo de prueba, por lo cual los casos de prueba producen un resultado binario: éxito ó falla.

Luego que se ha validado convenientemente el TS en formato JUnit (`JunitAccount`), el último paso de esta fase (de acuerdo a la Figura 3.8, pág. 69) implica derivar una versión equivalente del TS en formato MuJava, la cual será usada en la tercera fase del proceso. Así se generó la clase java `MuJavaAccount` (como instancia de la clase *TMuJavaFile* en el metamodelo de la Figura 3.7, pág. 68), y se pueden observar en la Tabla 3.5 los mismos métodos `testTS_1_1` y `testTS_75_1` en formato MuJava, los

Plantilla `testTS_1`	Plantilla `testTS_75`

public String testTS_1_1() {
try {
bank.Account obtained=null;
obtained =new bank.Account();
double result0=obtained.getBalance();
double arg1=(double) 50;
obtained.deposit(arg1);
return obtained.toString();
} catch (Exception ex) {
return ex.toString();
}
}
	public String testTS_75_1() {
	try {
	bank.Account obtained=null;
	obtained =new bank.Account();
	java.lang.String result0=obtained.toString();
	double arg1=(double) 50;
	obtained.deposit(arg1);
	double arg2=(double) 20;
	obtained.withdraw(arg2);
	return obtained.toString();
	} catch (Exception ex) {
	return ex.toString();
	}
	}

Tabla 3.5: Versión en formato MuJava de los Casos de Prueba para Account

cuales presentan un formato similar a los casos de prueba en formato JUnit, dado que son mínimas las variaciones entre ambos formatos: en MuJava no se agregaron restricciones en las que se incluya una sentencia *assert* (que actúe como oráculo), dado que estos métodos retornan un valor de tipo String (que será luego usado para efectuar las comparaciones). Por lo tanto la clase java `MuJavaAccount` representa el TS de Comportamiento de Componente para `Account`, que era el objetivo a ser alcanzado en esta fase inicial del proceso de compatibilidad.

En la siguiente sección se explicará la segunda fase del proceso, la cual se aplica cuando un componente de reemplazo candidato debe ser integrado en el sistema.

3.3. Compatibilidad de Interfaces

Cuando un componente es considerado como un potencial reemplazo para algún componente integrado en un sistema, se debería iniciar la fase de Compatibilidad de Interfaces. La evaluación que se realiza en esta fase toma ambos componentes para extraer sus interfaces y realizar una comparación a nivel sintáctico, de forma que se se pueda determinar si el componente de reemplazo ofrece los mismos servicios que el componente original, o en su defecto servicios con un grado razonable de equivalencia.

Para esta fase de Compatibilidad de Interfaces se ha definido un esquema de 4 niveles de equivalencia que permiten realizar una comparación para vincular servicios del componente original con aquellos del componente de reemplazo candidato. Estos

4 niveles han sido denominados: *exacto*, *casi-exacto*, *moderado*, *casi-moderado*; y los mismos expresan el grado de restricción que se debe cumplir al efectuar la vinculación de los servicios. Estos niveles se basan en condiciones específicas que consideran individualemente cada uno de los elementos que componen la signatura de los servicios (*retorno, nombre, parámetro, excepción*).

La Tabla 3.6 describe un conjunto de condiciones por cada elemento individual de las signaturas, las cuales se explican con mayor detalle en la Subsección 3.3.1. A su vez estas condiciones se deben considerar en presencia de las restantes, lo cual genera así los cuatro niveles de equivalencia para vincular servicios según se mencionó previamente. La Tabla 3.7 presenta tal combinación de condiciones individuales, mostrando todos los casos que se consideraron significativos para la comparación de interfaces de componentes. Se puede observar que por cada uno de los 4 niveles de clasificación se han identificado varios casos, que describen grados diferentes y específicos de equivalencia. La Figura 3.9 resume los casos particulares de equivalencia y se distinguen con un color distinto a la agrupación de los grados en cada nivel.

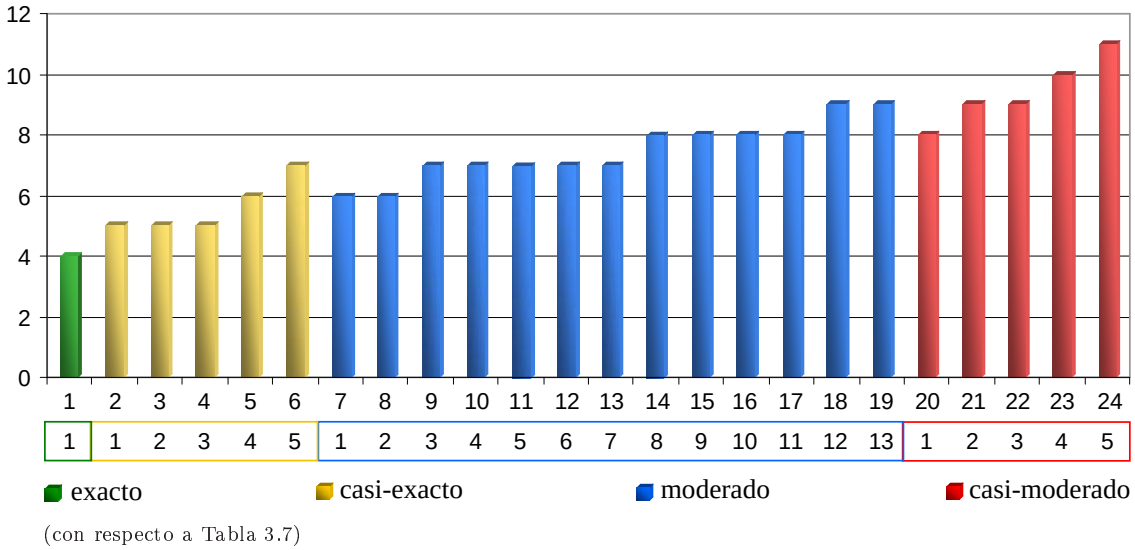
A continuación se describen los 4 niveles principales de equivalencia que permiten la comparación y vinculación de servicios (también distinguidos con el color correspondiente):

- EXACTO (*exact*). Dos servicios bajo comparación deben tener signaturas idénticas. Esto incluye el nombre del servicio (N1), el tipo de retorno (R1), y tanto para parámetros y excepciones: cantidad, tipo, y orden (P1,E1), lo cual implica satisfacer la siguiente conjunción de condiciones: R1∧N1∧P1∧E1, que además genera el valor de equivalencia 4 al sumar el valor 1 de cada condición.
  Ejemplo: para el caso de `Account` y `Cuenta`, el único servicio con equivalencia *exacta* es `toString` (como analizaremos en la Sección 3.3.2).

- CASI-EXACTO (*near-exact*). A este nivel se relaja el orden dentro de la lista para los parámetros y/o las excepciones (P2, E2), y para el nombre de los servicios se puede observar una equivalencia de substring (N2). De ello surgen 5 casos que generan valores de equivalencia entre 5 y 7, producto de la conjunción de condiciones simples, como se pueden observar en la Figura 3.9 y se detallan en la Tabla 3.7.
  Ejemplo: el servicio `getBalance` de `Account` tiene equivalencia *n_exact_3* con el servicio `getSaldo` de `Cuenta`, dado que coinciden en todo menos el nombre, para el cual existe equivalencia de substring ('get').

Elemento de Signatura	Condición	Descripción
Tipo de Retorno	R0	No Compatible
	R1	Idéntico
	R2	Equivalente (subtipos)
Nombre de Servicio	N1	Idéntico
	N2	Equivalente (substring)
	N3	Ignorado
Parámetros	P0	No Compatible
	P1	Igual cantidad, tipo y orden de parámetros en la lista
	P2	Igual cantidad y tipo para parámetros en la lista
	P3	Igual cantidad, pero tipo equivalente (subtipos) para algunos parámetros
Excepciones	E0	No Compatible
	E1	Igual cantidad, tipo y orden de excepciones en la lista
	E2	Igual cantidad y tipo en la lista de excepciones
	E3	Si lista de excepciones original no vacía, entonces no vacía lista en el candidato (sin importar los tipos)

Tabla 3.6: Condiciones de Equivalencia Sintáctica por Elemento de Signatura



(con respecto a Tabla 3.7)

Figura 3.9: Casos de Niveles de Equivalencia para Compatibilidad de Interfaces

- MODERADO (*soft*). Se consideran dos categorías de casos mutuamente exclusivas:
  1) similar al nivel anterior, aunque se ignora el nombre de los servicios (N3) y para las excepciones se relaja a solo identificar la existencia de alguna (E3);
  2) implica la equivalencia de subtipos para el retorno (R2) y los parámetros (P3), de lo cual a este nivel se requiere para los nombres de servicios: igualdad (N1) o equivalencia de substring (N2), y para las excepciones: igualdad de cantidad, tipos y quizá también orden (E1,E2). Esto genera 13 casos que generan valores de equivalencia entre 6 y 9, que se pueden observar en la Figura 3.9 y se detallan en la Tabla 3.7.
  Ejemplo: el servicio `deposit` de `Account` tiene equivalencia *soft_1* con los servicios `ingresar` y `retirar` de `Cuenta`, en este caso cumpliendo la categoría 1 al coincidir en todo menos el nombre, donde tampoco existe equivalencia de substring.

- CASI-MODERADO (*near-soft*). Similar a la categoría 1 del nivel previo, aunque considerando equivalencia de subtipos para el retorno (R2) y los parámetros (P3) a este nivel. Esto resulta en 5 casos que generan valores de equivalencia entre 8 y 11, que se pueden observar en la Figura 3.9 y se detallan en la Tabla 3.7.

El producto resultante de esta fase es un Mapeo de Compatibilidad donde cada correspondencia de servicios está caracterizada de acuerdo a los cuatro niveles de equivalencia presentados previamente. Un requisito para que el resultado de esta comparación no sea una incompatibilidad, implica que el componente candidato debería contener la interfaz ofrecida por el componente original, lo cual significa que el número de servicios ofrecidos por un componente candidato debería ser igual o mayor que los del componente original. Para ello, se ha reforzado que para todo servicio de un componente original debe existir al menos una correspondencia en el Mapeo de Compatibilidad. Si se reconociera una incompatibilidad para algún servicio original, el proceso requiere una decisión de un ingeniero de sosftware, que puede involucrar el suministro manual de una correspondencia para tal servicio que permita continuar con el proceso, o bien simplemente detener el proceso concluyendo la incompatibilidad del componente candidato.

En un framework orientado a objetos como Java, existe un conjunto de métodos que se heredan de la clase *Object* [Gosling y otros, 2005], los cuales están siempre presentes a menos que no se considere la herencia en el proceso de evaluación. En algunos casos esos métodos pueden servir de ayuda para encontrar correspondencias cuando algunos de

Nivel	Grado	Caso	Elementos de Signatura‡				Valor
			R	N	P	E	
■ Exacto	exact	01	1	1	1	1	4
■ Casi-Exacto (*near-exact)*	n_exact_1	02	1	1	1	2	5
	n_exact_2	03	1	1	2	1	5
	n_exact_3	04	1	2	1	1	5
	n_exact_4	05	1	2	2	1	6
	n_exact_5	06	1	2	2	2	7
■ Moderado (*soft)*	soft_1	07	1	3	1	1	6
	soft_2	08	1	1	3	1	6
	soft_3	09	1	1	2	3	7
	soft_4	10	1	3	2	1	7
	soft_5	11	1	1	3	2	7
	soft_6	12	1	2	3	1	7
	soft_7	13	2	1	3	1	7
	soft_8	14	1	2	2	3	8
	soft_9	15	1	2	3	2	8
	soft_10	16	2	1	3	2	8
	soft_11	17	2	2	3	1	8
	soft_12	18	1	3	2	3	9
	soft_13	19	2	2	3	2	9
■ Casi-Moderado (*near-soft)*	n_soft_1	20	1	1	3	3	8
	n_soft_2	21	2	1	3	3	9
	n_soft_3	22	1	2	3	3	9
	n_soft_4	23	2	2	3	3	10
	n_soft_5	24	2	3	3	3	11

‡ con respecto a la Tabla 3.6

Tabla 3.7: Casos de Niveles de Equivalencia para Compatibilidad de Interfaces

ellos han sido convenientemente sobreescritos, sin embargo, generalmente tales métodos no resultan significativos para participar en una comparación. Así, una opción válida podría ser la omisión de tales métodos en un primer intento, y luego en el caso de no encontrar una correspondencia para algún servicio, incorporar los métodos de la clase *Object* en la comparación para observar un cambio en los resultados.

El procedimiento para la Compatibilidad de Interfaces construye el Mapeo de Compatibilidad, vinculando a cada servicio s_C de un componente original C, con una lista conteniendo los servicios de un componente de reemplazo K, que son compatibles con s_C. Por ejemplo, sea C con tres servicios s_{Ci}, $1 \leq i \leq 3$, y K con cinco servicios s_{Kj}, $1 \leq j \leq 5$. Luego de aplicar el procedimiento se retorna un Mapeo de Compatibilidad que podría adoptar la siguiente forma:

$$\{(s_{C1},\{(\text{n\_ex},s_{K1}),(\text{soft},s_{K2}),(\text{n\_soft},s_{K5})\}),(s_{C2},\{(\text{ex},s_{K2}),(\text{soft},s_{K4}),(\text{soft},s_{K5})\}),(s_{C3},\{(\text{soft},s_{K3})\})\}$$

El procedimiento efectúa la búsqueda de correspondencias iniciando con un nivel de equivalencia de mayor restricción, para luego continuar con los más debiles (es decir, de una equivalencia *exacta* a una *casi-moderada*). Es muy importante identificar correspondencias de mayor nivel de equivalencia porque el resultado de esta fase significa un conocimiento pre-analizado de los componentes en evaluación, que luego se utiliza como base para que en la tercera fase del proceso, se pueda finalmente llegar a un resultado conclusivo sobre compatibilidad. Además, cuanto mayor es el nivel de equivalencia encontrado en las correspondencias, mayor será la reducción de esfuerzo de computación en la tercera fase, beneficiando de esta manera el rendimiento o eficiencia del proceso. Esto se debe a que a mayores niveles de equivalencia, la cantidad de correspondencias es generalmente menor – por ejemplo en el caso de una equivalencia *exacta* sólo puede existir una única correspondencia. Esto se explica con mayor profundidad en la tercera fase, por medio del ejemplo práctico.

Como se puede observar en la Tabla 3.7, cada uno de los casos particulares de equivalencia determinan un valor específico (por ejemplo la equivalencia *exacta* es igual a 4), y por lo tanto al construir el Mapeo de Compatibilidad se puede determinar un valor totalizado para definir la Compatibilidad de Interfaces. Este valor totalizado puede inicialmente considerar sólo las equivalencias de mayor nivel, y así determinar lo que se denomina "*distancia sintáctica*" entre el componente original C y el componente de reemplazo K. A continuación se muestra la fórmula correspondiente para el cálculo de *distancia sintáctica*:

$$syntDist(C,K) = \frac{\sum_{i=1}^{N} Min(s_{Ci}, MapComp(C,K))}{N * 4} - 1 \quad (3.1)$$

donde

N: tamaño de la interfaz de C
Min: función que devuelve el mínimo valor (mejor equivalencia obtenida de s_{Ci})
$MapComp$: Mapeo de Compatibilidad entre C y K

En el caso que todos los servicios del componente C lograran una equivalencia *exacta* (que es igual al valor 4) con algún servicio del componente K, la sumatoria del numerador en la fórmula sería igual al valor del denominador ($N*4$), con lo cual la *distancia sintáctica*

entre C y K sería cero. Esto no significa que los componentes sean idénticos, sino que al menos la interfaz de C está incluída en la interfaz de K. El componente candidato podría tener una interfaz extendida, ofreciendo servicios adicionales, lo que fuerza a cerciorarse que no producirán algún conflicto de integración, y para ello se requiere del análisis de comportamiento que se realiza en la tercera fase del proceso.

3.3.1. Condiciones para Equivalencia Sintáctica de Servicios

Sea C un componente original y K su componente de reemplazo candidato, para todos los servicios (s_C,s_K), donde s_C pertenece a la interfaz de C y s_K pertenece a la interfaz de K, se detallan en esta subsección el conjunto de condiciones individuales para la probable equivalencia entre s_C y s_K. En la Tabla 3.8 se presentan nuevamente los niveles individuales de equivalencia mostrados en la Tabla 3.6 (de la sección previa, pág. 75), indicando en particular para los parámetros y excepciones el desglose y combinación de condiciones más simples, presentadas en la Tabla 3.9.

Elemento de Signatura	Nivel†	Condiciones Simples‡
Tipo de Retorno	R0	No Compatible
	R1	Idéntico
	R2	Equivalente (subtipos)
Nombre de Servicio	N1	Idéntico
	N2	Equivalente (substring)
	N3	Ignorado
Parámetros	P0	No Compatible
	P1	`Pc1` $\wedge$ `Pc2` $\wedge$ `Pc3`
	P2	`Pc1` $\wedge$ `Pc2` $\wedge \neg$`Pc3`
	P3	`Pc1` $\wedge$ `Pc2.1`
Excepciones	E0	No Compatible
	E1	`Ec1` $\wedge$ `Ec2`
	E2	`Ec1` $\wedge \neg$`Ec2`
	E3	si lista de excepciones original no vacía, entonces no vacía lista en el candidato (sin importar los tipos)

†con respecto a la Tabla 3.6 (de la sección previa, pág. 75)

‡con respecto a la Tabla 3.9 para el caso de parámetros y excepciones

Tabla 3.8: Nivel de Equivalencia Sintáctica por Elemento de Signatura

Elemento de Signatura	Condición	Descripción
Parámetros	Pc1	igual cantidad de parámetros
	Pc2	igual tipo para cada parámetro en la lista
	Pc2.1	tipo equivalente para algunos parámetros (subtipos)
	Pc3	orden idéntico en la lista de parámetros
Excepciones	Ec1	igual cantidad de excepciones y tipos idénticos para cada excepción en la lista
	Ec2	orden idéntico en la lista de excepciones

Tabla 3.9: Condiciones Simples de Equivalencia Sintáctica por Elemento de Signatura

Tipo de Retorno (Tipo de Servicio):

- *Condición R1.* El tipo de retorno de ambos servicios es el mismo:
$$\texttt{typeOf(s}_C\texttt{)} = \texttt{typeOf(s}_K\texttt{)}$$
Siendo `typeOf(x)` una función que retorna un tipo, donde `x` puede ser un servicio, un parámetro, o una excepción.

- *Condición R2.* El tipo de retorno de ambos servicios es equivalente.
$$\texttt{typeOf(s}_C\texttt{)} \approx \texttt{typeOf(s}_K\texttt{)}$$
La equivalencia de los tipos de datos ha sido establecida en función de la relación de subsumción conocida como *subtipificación* (se escribe <:) [Zaremski y Wing, 1997; Gosling y otros, 2005], y en particular ha sido definida para tipos primitivos (*built-in*) en nuestro enfoque, de acuerdo a la relación *directa* de subtipos (se escribe $<_1$) del lenguaje Java [Gosling y otros, 2005], que se muestra en la Tabla 3.10.

1° Tipo		2° Tipo
byte	$<_1$	short
short	$<_1$	int
int	$<_1$	long
long	$<_1$	float
float	$<_1$	double

Tabla 3.10: Subtipificación Directa para Tipos Primitivos

Así para las signaturas de los servicios se consideran relaciones de subtipos como se muestra en la Tabla 3.11, donde los tipos en $\texttt{s}_K$ deben al menos tener tanta precisión como los tipos en $\texttt{s}_C$. Por ejemplo si el servicio $\texttt{s}_C$ incluye un tipo *int*, entonces un servicio correpondiente $\texttt{s}_K$ no puede tener una presición menor como *short* o *byte* (entre los tipos numéricos).

Tipo en s_C	Tipo en s_K
char	string
byte	short, int, long, o double
short	int, long, o double
int	long, o double
long	double

Tabla 3.11: Equivalencia de Subtipos para Servicios

Nombre de Servicio:

- *Condición N1.* El nombre de ambos servicios es idéntico:

$$\texttt{nameOf}(s_C) = \texttt{nameOf}(s_K)$$

- *Condición N2.* El nombre de ambos servicios es equivalente.

$$\texttt{nameOf}(s_C) \approx \texttt{nameOf}(s_K)$$

  La equivalencia de nombres de servicios implica intentar encontrar similitud en substring. Por ejemplo, en los servicios `getBalance` y `getSaldo` del ejemplo, existe una similitud en el substring "get", que aunque no es altamente significativa, al menos permite distinguir un tipo de método accesor de otro tipo de métodos. En el Capítulo 4 se muestra un caso de estudio donde esta equivalencia de substring resulta significativa, por ejemplo para los servicios `putInBuffer` y `addToBuffer`, donde se puede reconocer una similitud en el substring "Buffer".

Parámetros:

- *Condición Pc1.* El número de parámetros en ambos servicios es idéntico:

$$\texttt{size}(\texttt{params}(s_C)) = \texttt{size}(\texttt{params}(s_K)$$

- *Condición Pc2.* Los tipos de los parámetros en ambos servicios son idénticos:

$$\begin{array}{c}\texttt{size(params(s}_C\texttt{))} = \texttt{size(params(s}_K\texttt{))} \wedge (\ \forall\ \texttt{i,j,x,y}\ /\\ 1 \leq \texttt{i} < \texttt{j} \leq \texttt{size(params(s}_C\texttt{))} \wedge \texttt{x} \neq \texttt{y} \wedge 1 \leq \texttt{x,y} \leq \texttt{size(params(s}_K\texttt{))} \bullet\\ \texttt{typeOf(parms(s}_C\texttt{)[i])} = \texttt{typeOf(parms(s}_K\texttt{)[x])} \wedge\\ \texttt{typeOf(parms(s}_C\texttt{)[j])} = \texttt{typeOf(parms(s}_K\texttt{)[y])}\ )\end{array}$$

- *Condición Pc2.1.* Los tipos de los parámetros en ambos servicios son al menos equivalentes.
  La equivalencia se basa en aquella definida en la Condición Rc2.

$$\begin{array}{c}\texttt{size(params(s}_C\texttt{))} = \texttt{size(params(s}_K\texttt{))} \wedge (\ \forall\ \texttt{i,j,x,y}\ /\\ 1 \leq \texttt{i} < \texttt{j} \leq \texttt{size(params(s}_C\texttt{))} \wedge \texttt{x} \neq \texttt{y} \wedge 1 \leq \texttt{x,y} \leq \texttt{size(params(s}_K\texttt{))} \bullet\\ (\ \texttt{typeOf(parms(s}_C\texttt{)[i])} \approx \texttt{typeOf(parms(s}_K\texttt{)[x])} \vee\\ \texttt{typeOf(parms(s}_C\texttt{)[i])} = \texttt{typeOf(parms(s}_K\texttt{)[x])}\ ) \wedge\\ (\ \texttt{typeOf(parms(s}_C\texttt{)[j])} \approx \texttt{typeOf(parms(s}_K\texttt{)[y])} \vee\\ \texttt{typeOf(parms(s}_C\texttt{)[j])} = \texttt{typeOf(parms(s}_K\texttt{)[y])}\ )\end{array}$$

- *Condición Pc3.* Ambos servicios tienen el mismo orden en la lista de parámetros. Esto depende de las Condiciones Pc1 y Pc2.

$$\begin{array}{c}\texttt{size(params(s}_C\texttt{))} = \texttt{size(params(s}_K\texttt{))} \wedge (\ \forall\ 1 \leq \texttt{i} \leq \texttt{size(params(s}_C\texttt{))}\ /\\ \texttt{typeOf(parms(s}_C\texttt{)[i])} = \texttt{typeOf(parms(s}_K\texttt{)[i])}\ )\end{array}$$

Excepciones:

- *Condición Ec1.* El número de excepciones en ambos servicios es igual, y el tipo de cada par correspondiente de excepciones es idéntico:

$$\begin{array}{c}\texttt{size(exceptions(s}_C\texttt{))} = \texttt{size(exceptions(s}_K\texttt{)} \wedge\\ (\ \forall\ \texttt{i,j,x,y}\ /\ 1 \leq \texttt{i} < \texttt{j} \leq \texttt{size(exceptions(s}_C\texttt{))} \wedge\\ \texttt{x} \neq \texttt{y} \wedge 1 \leq \texttt{x,y} \leq \texttt{size(exceptions(s}_K\texttt{))} \bullet\\ \texttt{typeOf(exceptions(s}_C\texttt{)[i])} = \texttt{typeOf(exceptions(s}_K\texttt{)[x])} \wedge\\ \texttt{typeOf(exceptions(s}_C\texttt{)[j])} = \texttt{typeOf(exceptions(s}_K\texttt{)[y])}\ )\ )\end{array}$$

- *Condición Ec2.* Ambos servicios tienen el mismo orden en la lista de excepciones. Esto depende de la Condición Ec1.

$$\begin{array}{c}\texttt{size(exceptions(s}_C\texttt{))} = \texttt{size(exceptions(s}_K\texttt{))} \wedge\\ (\ \forall\ 1 \leq \texttt{i} \leq \texttt{size(exceptions(s}_C\texttt{))}\ /\\ \texttt{typeOf(exceptions(s}_C\texttt{)[i])} = \texttt{typeOf(exceptions(s}_K\texttt{)[i])}\ )\end{array}$$

- *Condición E3.* Si la lista de excepciones para el servicio s_C no es vacía, entonces tampoco puede ser vacía la lista de excepciones para el servicio s_K:

$$\texttt{size(exceptions(s}_C\texttt{))} \neq 0 \Rightarrow \texttt{size(exceptions(s}_K\texttt{))} \neq 0$$

Luego de una visión detallada de las condiciones para las correspondencias syntácticas de interfaces, a continuación se puede observar cómo la fase de Compatibilidad de Interfaces se lleva a cabo por medio del caso práctico presentado en una sección previa.

3.3.2. Correspondencia de Interfaces entre Account y Cuenta

Todas las condiciones, grados y niveles de equivalencia para interfaces descritos en la sección previa, han generado un módulo adicional que ha sido implementado por el doctorando en la herramienta *testooj*, donde nuevamente se utiliza el metamodelo de la Figura 3.7 (de la sección previa, pág. 68) como capa subyacente, para efectuar la extracción de las interfaces y proceder a su tratamiento. Así la automatización de la fase permite que el análisis comparativo sea efectuado de manera rigurosa, lo cual facilita identificar el origen de los resultados que se obtienen.

Luego del análisis de Compatibilidad de Interfaces en el cual se construye un Mapeo de Compatibilidad, se presentan los datos en un cuadro donde por cada servicio del componente original, se listan los servicios del componente candidato con los cuales se encontró una correspondencia, ordenados por el nivel y grado específico de equivalencia alcanzado. El formato con el que se presentan los servicios candidatos se corresponde con la Tabla 3.7 (de la sección previa, pág. 77) que describe los casos de niveles de equivalencia que se consideran significativos para la comparación de interfaces de componentes:

[Caso, GradoEq, Signatura, RX, NX, PX, EX]

donde:

Caso: nivel de equivalencia numérico del servicio candidato
GradoEq: grado de equivalencia descriptivo del servicio candidato
(de acuerdo a la clasificación de 4 niveles)
Signatura: signatura del servicio candidato
RX: nivel de equivalencia en el tipo de retorno
NX: nivel de equivalencia en el nombre del servicio
PX: nivel de equivalencia en los parámetros
EX: nivel de equivalencia en las excepciones

Al aplicar los procedimientos de Compatibilidad de Interfaces a los componentes `Account` y `Cuenta`, la herramienta *testooj* presenta los resultados del Mapeo de Compatibilidad generado, en un cuadro similar al mostrado en la Tabla 3.12. Un resumen de estos resultados se presenta en la Tabla 3.13, desde el cual se puede aplicar la Fórmula 3.1 (de la Sección 3.3, pág 78) para calcular la *distancia sintáctica* entre `Account` y `Cuenta`, la cual sería: (28/20)-1=0,4.

bank.Account	**banco.Cuenta**	
double getBalance()	[4, n_exact_3, double getSaldo(), R1, N2, P1, E1]	
java.lang.String toString()	[1, exact, java.lang.String toString(), R1, N1, P1, E1]	
void deposit(double x1)	[7, soft_1, void ingresar(double x1), R1, N3, P1, E1]	[7, soft_1, void retirar(double x1), R1, N3, P1, E1]
void withdraw(double x1)	[7, soft_3, void ingresar(double x1), R1, N3, P1, E1]	[7, soft_1, public void retirar(double x1), R1, N3, P1, E1]
void transfer(java.lang.String x1, double x2)	[10, soft_4, void transferir(double x1, java.lang.String x2), R1, N3, P2, E1]	

Tabla 3.12: Compatibilidad de Interfaces entre los componentes Account y Cuenta

Exacta	Casi-Exacta	Moderada	Casi-Moderada	(Cantidad)	Servicios de Account	Mejor Valor
■ 1				(1)	`toString`	4
	■ 1			(1)	`getBalance`	5
		■ 1		(1)	`transfer`	7
		■ 2		(2)	`deposit, withdraw`	6
Total servicios de Account				5	Total Compatibilidad	28§

§ Mejor Compatibilidad = 20

Tabla 3.13: Resumen de Compatibilidad de Interfaces para Account-Cuenta

De ambas tablas se desprenden algunos aspectos interesantes que se describen a continuación:

- Se ha encontrado una correspondencia para todos los servicios de `Account`, que de otra manera hubiera llevado el proceso a su fin, o bien hubiera necesitado la generación manual de una correspondencia para ser utilizada en la siguiente fase del proceso.

- Un nivel alto de compatibilidad es importante para la tercera fase del proceso como se explicará mas tarde. Este fue el caso con dos los primeros servicios mostrados en la Tabla 3.13, y aunque el servicio `transfer` obtuvo una equivalencia *moderada*, ha sido importante que se identificara una única correspondencia, dado que a mayor cantidad de correspondencias el procedimiento para la siguiente fase se torna más complejo.

- El servicio `getBalance` obtuvo una correspondencia a nivel *casi-exacto*, donde la equivalencia de substring permitió distinguir un caso de nivel mayor que *moderado*. Esto se reconoce en la Tabla 3.12 con el identificador (N2), para la correspondencia con el servicio `getSaldo` de `Cuenta`.

- Dos servicios obtuvieron una equivalencia *moderada* con el mismo caso particular de correspondencia, y que generó además una vinculación cruzada. Estos son los servicios `deposit` y `withdraw` que han sido vinculados a los servicios `ingresar` y `retirar` del componente `Cuenta`. Se puede observar en la Tabla 3.12 que el elemento que afectó la comparación fue el nombre de los servicios, el cual debió ser ignorado – marcado con (N3). Estas correspondencias serán las que afecten al procedimiento de la tercera fase del proceso.

El Mapeo de Compatibilidad obtenido en esta fase permite la oportunidad de descubrir una potencial compatibilidad de componentes al proveer información para la siguiente fase que involucra la evaluación semántica de compatibilidad basada en pruebas.

3.4. Compatibilidad de Comportamiento

Esta fase no solamente puede generar una diferenciación entre los servicios sintácticamente similares, sino que principalmente asegura que las correspondencias de interfaces también sean válidas a nivel semántico. Esto significa que el propósito es encontrar servicios de un componente de reemplazo candidato que expongan un comportamiento similar con respecto al componente original de referencia. En nuestro enfoque, esto implica ejercitar el TS de Comportamiento de Componente, generado en la primera fase del proceso, contra el componente de reemplazo.

La automatización de esta fase está basada en la información de correspondencia sintáctica surgida del análisis de Compatibilidad de Interfaces, la cual se utiliza para construir wrappers para el componente de reemplazo candidato. Cada wrapper será una clase que puede reemplazar al componente original de referencia, dado que incluye la misma interfaz y aún el mismo nombre de clase. Un wrapper posee por lo tanto el comportamiento establecido por el *patrón de diseño adaptador* [Gamma y otros, 1995], simplemente recibiendo los requerimientos del cliente y reenviándolos adecuadamente al componente candidato. La cantidad de wrappers se establece en función de las combinaciones de correspondencias de servicios. En vez de simplemente realizar una combinación ciega, es posible obtener un conjunto más reducido a través de la evaluación sintáctica de la fase previa.

El enfoque de wrappers hace uso así de aspectos provenientes de la técnica de *mutación de interfaces* [Gosh y Mathur, 2001; Delamaro y otros, 2001] aplicando operadores de mutación para cambiar invocaciones a servicios y también para cambiar valores de los argumentos para los parámetros. El primer operador se aplica a través del Mapeo de Compatibilidad de servicios, mientras que el segundo operador se aplica mediante la variación de los argumentos en los parámetros que poseen el mismo tipo (en una invocación a un servicio del componente candidato, luego de haber fijado una correspondencia particular a partir del Mapeo de Compatibilidad).

La Figura 3.10 muestra cómo para cada servicio del componente original C, sus correspondencias hacia el componente candidato K han sido reacomodadas por medio de una estructura de árbol. Cada nivel en el árbol implica correspondencias para un servicio distinto de la interfaz de C. En el caso que un servicio incluya una lista de parámetros de tamaño mayor que uno, cada combinación de la correspondencia de tipos producirá un wrapper, lo cual implica otro nodo hijo en el árbol (es decir, una nueva rama). Esto puede observarse en la Figura 3.10, con el servicio s_{C3} el cual incluye una lista de parámetros de tamaño L.

Cada camino del árbol (de la raíz a un nodo hoja) representa un wrapper diferente a ser generado, donde el nivel de hojas describe la cantidad total de wrappers, que es el resultado de las correspondencias obtenidas por cada servicio del componente C. Por ejemplo en la Figura 3.10, el servicio s_{Cj+1} (entre otros) tiene una correspondencia con V servicios del componente K, lo cual genera que se agreguen V ramas bajo cada hoja actual del árbol.

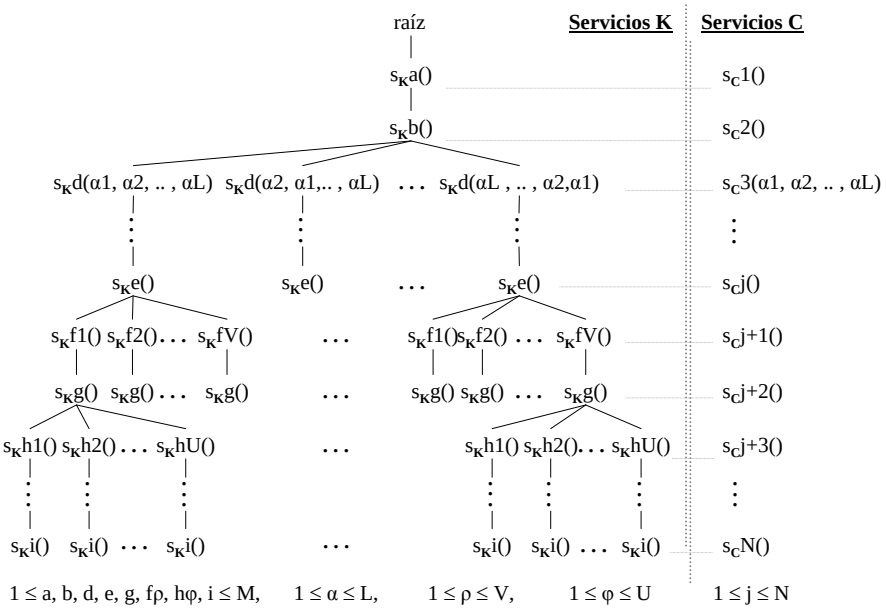


Figura 3.10: Generación de wrappers a través de la construcción de un árbol de servicios

No obstante, la cantidad de correspondencias puede ser reducida al tomar el nivel de equivalencia más alto obtenido en el análisis de Compatibilidad de Interfaces de la segunda fase del proceso. Esto se puede observar en uno de los casos prácticos desarrollados en el Capítulo 4 donde uno de los servicios (`clear`), resultó con una equivalencia *exacta*, y además 21 correspondencias con un nivel más bajo de equivalencia (ver la Sección 4.2), con lo cual si se toma sólo esa única correspondencia más alta y se omiten las restantes de nivel más bajo, este servicio no produce una rama adicional en el árbol – por ejemplo, los servicios s_{C1} y s_{C2} (entre otros) en la Figura 3.10.

En resumen, la cantidad total de wrappers es el resultado de un producto de todos los servicios del componente C que tienen más de una correspondencia a servicios del componente K y aquellos que tienen correspondencias en la lista de parámetros. A continuación se puede observar la fórmula correspondiente para el cálculo de tamaño del conjunto de wrappers W:

$$size(W) = \prod_{i=1}^{N} \sum_{j=1}^{V_i} \prod_{t=1}^{T_i} p_t! \qquad (3.2)$$

donde

N: tamaño de la interfaz de C
M: tamaño de la interfaz de K
V_i: cantidad de correspondencias de s_{Ci} hacia la interfaz de K, $1 \leq V_i \leq M$
L_i: tamaño de la lista de parámetros para s_{Ci}
T_i: cantidad de diferentes tipos en la lista de parámetros para s_{Ci}, $0 \leq T_i \leq L_i$
p_t: cantidad de ocurrencias para un tipo en la lista de parámetros para s_{Ci}, $0 \leq p_t \leq L_i$
$p_t!$: permutaciones de las ocurrencias para un tipo de parámetro en el servicio s_{Ci}

Dado que el tamaño de W podría ser bastante grande, lo cual puede decrementar considerablemente el rendimiento de la fase actual, se podría considerar una opción diferente. La fase previa que analiza la Compatibilidad de Interfaces provee un conocimiento importante sobre las correspondencias de servicios, al resaltar el propósito probable de diferentes servicios (es decir, su comportamiento). Esto básicamente se refiere a la relación entre un nombre de servicio y su supuesto propósito como parte de la funcionalidad completa provista por el componente en evaluación. Así, para algunos servicios podría resultar muy claro cuál debería ser la correspondencia apropiada.

Por lo tanto un ingeniero de software podría tomar la decisión de establecer manualmente todas las correspodencias, para así construir un único wrapper, para lo cual la herramienta *testooj* provee facilidades adicionales para tal fin. Si fuera el caso que ese wrapper no generara un resultado exitoso, se podría aplicar alguna otra correspondencia, o bien tomar la decisión de cambiar a una generación automática de un conjunto mas grande de wrappers.

Luego de construir el conjunto W la Compatibilidad de Comportamiento puede proceder tomando cada wrapper como el componente bajo prueba y ejecutando el TS de Comportamiento de Componente generado en la primera fase del proceso. La Figura 3.11 muestra tres pasos del proceso: primero cómo se obtienen los resultados de las pruebas del componente original de referencia C, segundo cómo los wrappers del componente candidato K se prueban contra el TS generado para C, y tercero cómo los resultados de

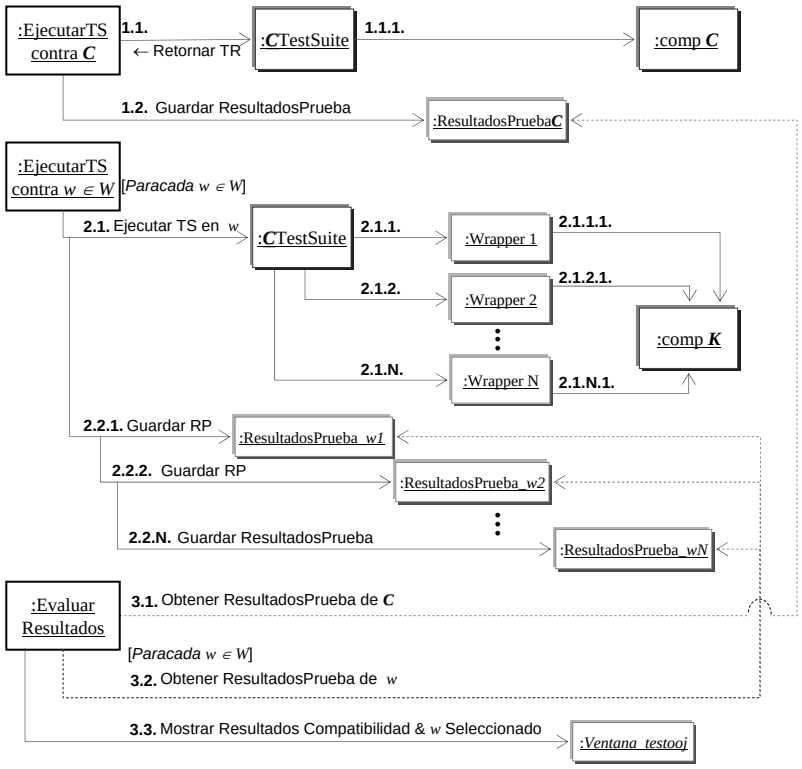


Figura 3.11: Ejecución del TS contra C y los wrappers de K, y evaluación de sus resultados

los wrappers probados son comparados con aquellos obtenidos para C. La evaluación de los casos de prueba se realiza por medio de los valores de retorno, los cuales tienen formato String. La comparación realizada genera por lo tanto una evaluación de cada caso de prueba con resultado binario: éxito ó fallo. El porcentaje de pruebas exitosas de cada wrapper determina su aceptación o rechazo, es decir si se permite o no que sobreviva el wrapper (como un caso de mutación). Cuanto mayor sea la cantidad de wrappers muertos, mejor es el resultado final, dado que facilita la toma de decisiones sobre la compatibilidad del componente en evaluación.

A continuación se describe cómo se realiza la fase de Compatibilidad de Comportamiento para el ejemplo que se ha desarrollado a través de las secciones previas.

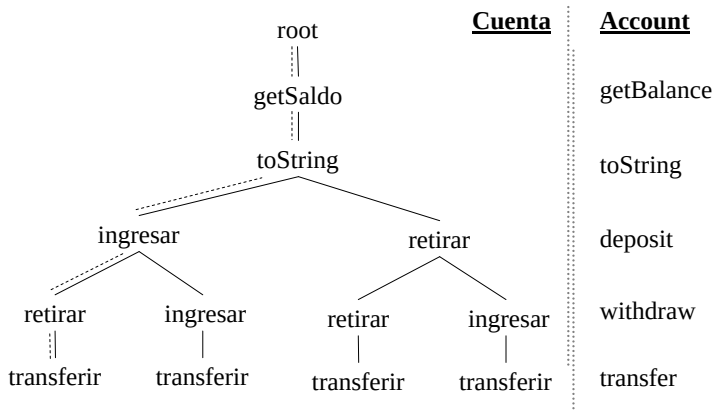


Figura 3.12: Generación de wrappers a través de la construcción de un árbol de servicios

3.4.1. Compatibilidad Semántica entre Account y Cuenta

Para iniciar el análisis de Compatibilidad de Comportamiento entre `Account` y `Cuenta` se necesita construir el conjunto de wrappers W de acuerdo al Mapeo de Compatibilidad generado en la segunda fase. Se ha considerado aquí tomar el nivel más alto de equivalencia posible para la construcción de los wrappers en este caso práctico, omitiendo por lo tanto cualquier otra correspondencia existente de menor nivel para servicios con diferentes niveles de equivalencia.

La Figura 3.12 muestra el árbol que se genera previo a la construcción de los wrappers, donde la cantidad de nodos en el nivel de las hojas es 4 (es decir el tamaño de W), y esto proviene de aquellos servicios que obtuvieron más de una correspondencia en el Mapeo de Compatibilidad. Dado que dos servicios de `Account` (`deposit` y `withdraw`) obtuvieron cada uno, una correspondencia con 2 servicios de `Cuenta`, como se muestra en la Tabla 3.13 (Sección 3.3.2, pág. 84), al aplicar la Fórmula 3.2 (pág. 88) se puede calcular el tamaño del conjunto W, que en este caso sería $2 * 2 = 4$.

La parte superior de la Figura 3.13 muestra el wrapper que correponde al primer camino (de la raíz al nodo hoja más a la izquierda) en el árbol de la Figura 3.12 (marcado con lineas punteadas), y que representa las verdaderas correspondencias de servicios para `Account` y `Cuenta`. Por otro lado, la parte inferior la Figura 3.13 muestra el wrapper que corresponde al cuarto camino en el árbol de la Figura 3.12, y que representa una versión defectuosa pero permite identificar las diferencias entre ambos wrappers.

wrappers/_1/bank/Account.java

```
package bank;

public class Account {
   protected banco.Cuenta candidate=new banco.Cuenta();

   public double getBalance () {
      return candidate.getSaldo();
   }
   public java.lang.String toString () {
      return candidate.toString();
   }
   public void deposit (double arg1) {
      candidate.ingresar(arg1);
   }
   public void withdraw (double arg1) {
      candidate.retirar(arg1);
   }
   public void transfer (java.lang.String arg1, double arg2) {
      candidate.transferir(arg2, arg1);
   }
}
```

wrappers/_4/bank/Account.java

```
package bank;

public class Account {
   protected banco.Cuenta candidate=new banco.Cuenta();

   public double getBalance () {
      return candidate.getSaldo();
   }
   public java.lang.String toString () {
      return candidate.toString();
   }
   public void deposit (double arg1) {
      candidate.retirar(arg1);
   }
   public void withdraw (double arg1) {
      candidate.ingresar(arg1);
   }
   public void transfer (java.lang.String arg1, double arg2) {
      candidate.transferir(arg2, arg1);
   }
}
```

Figura 3.13: Wrappers de Account para el componente Cuenta

El siguiente paso es ejecutar el TS de Comportamiento de Componente guardado en el fichero `MuJavaAccount` sobre cada wrapper del conjunto W para así evaluar la compatibilidad semántica. Para esto se utiliza una opción de la herramienta *testooj*, añadida durante el desarrollo de esta tesis, que permite tomar el fichero de pruebas e iterar a través del conjunto de wrappers, para luego realizar el análisis de resultados al comparar con el componente original `Account`. A los wrappers que han tenido un alto porcentaje de casos de prueba fallidos, se los considera como rechazados y por lo tanto se corresponden con mutantes muertos, dado que fueron generados por la aplicación de la técnica de *mutación de interfaces*.

En la Tabla 3.14 se muestra un resumen de los resultados donde solamente un wrapper pasó exitosamente todas las pruebas. Esto significa que sólo un wrapper puede sobrevivir (como caso de mutación), lo cual facilita la toma de decisiones en lo concerniente a aceptar o descartar el componente de reemplazo candidato, es decir `Cuenta` en este ejemplo.

	Casos de Prueba		
Wrapper	Exitosas	Fallidas	% de Éxito
1	228	0	100
2	26	202	11
3	124	104	54
4	62	166	27

Tabla 3.14: Resultados de ejecutar el TS de Account sobre Cuenta

El wrapper que sobrevivió no solo ayuda a descubrir la compatibilidad entre `Account` y `Cuenta`, sino que además representa el artefacto que un ingeniero de software require para ajustar al componente candidato (`Cuenta` en este ejemplo) para que pueda ser efectivamente ensamblado en el sistema.

3.5. Síntesis del Proceso

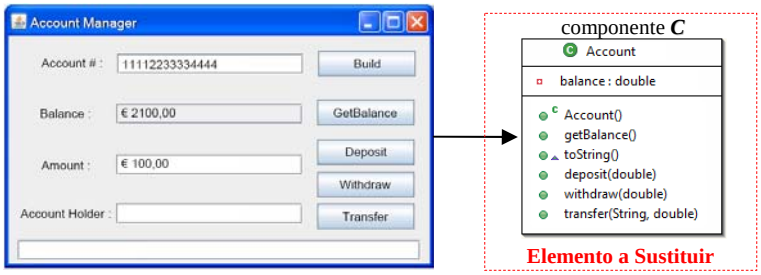
Como se ha explicado en esta sección nuestra propuesta es un proceso que se aplica durante el mantenimiento de un sistema, en el cual se necesita reemplazar uno de sus componentes, y por lo tanto se requiere evaluar si algún componente candidato puede actuar de sustituto en forma segura, es decir sin afectar el comportamiento de los componentes clientes de aquel componente que se está reemplazando, y por ende no

afectar la estabilidad del sistema.

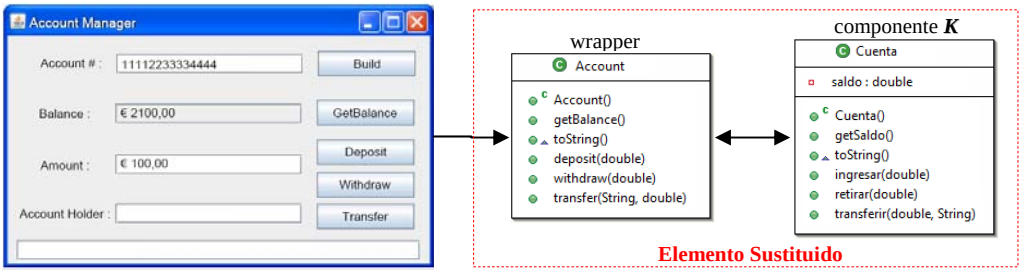
El proceso de compatibilidad que se ha propuesto, evalúa un componente original y otro de reemplazo, tanto a nivel sintáctico como semántico. A nivel sintáctico se analiza si la interfaz del componente original está contenida (con un grado razonable de equivalencia) en la interfaz del componente de reemplazo, para asegurar la interacción sintáctica de los clientes del componente original, con el componente de reemplazo. Luego a nivel semántico se aplican estrategias de pruebas para determinar la compatibilidad de comportamiento de los componentes. Para ello se ha utilizado la idea de la técnica de Pruebas Back-to-Back, la cual juzga la correctitud de una nueva implementación en función de generar un TS para una implementación anterior (de referencia), y ejecutar ese TS contra ambas implementaciones para comparar los resultados [Vouk y otros, 1987; Shimeall y Leveson, 1991; Edwards, 2001; Veenendaal, 2006]. En particular para el proceso que se ha propuesto, la intención no es encontrar fallos, sino comprobar la compatibilidad de comportamiento para ambos componentes, y por ello el TS que se genera, está diseñado en base a criterios de cobertura específicos que permitan describir el comportamiento del componente original. Luego en la ejecución del TS contra el componente de reemplazo, se aplica todo el conocimiento descubierto durante la evaluación sintáctica, para desarrollar los ajustes necesarios ante una equivalencia no exacta de las interfaces de ambos componentes.

Dado que pueden surgir diversos candidatos de reemplazo para el componente a ser sustituído, el proceso de evaluación que se ha propuesto además suministra un conocimiento que permite identificar la mejor elección entre el conjunto de componentes candidatos, y por lo tanto provee el soporte necesario para un enfoque de *selección de componentes*.

Gráficamente y aplicando el ejemplo con el que se ha ilustrado el proceso, el reemplazo del componente original `Account` por el componente candidato `Cuenta`, puede observarse como en la Figura 3.14.



(a) Sistema con componente Original



(b) Sistema luego de la Sustitución

Figura 3.14: Ilustración de una Sustitución en el ejemplo del Sistema Bancario

Capítulo 4

Casos Prácticos

4.1. Introducción

En este capítulo se desarrollarán algunos casos prácticos para mostrar resultados concretos de la aplicabilidad del Proceso de Evaluación para la Sustitución de Componentes Software que se ha presentado en el capítulo anterior. El primer caso práctico que se presenta en la Sección 4.2 involucra una calculadora Java mientras que el segundo caso práctico que se desarrolla en la Sección 4.4 presenta un componente para ser integrado en un parser o compilador. En ambos casos prácticos se han descargado los componentes de sitios web públicos, para permitir que se pueda cotejar el origen de los resultados obtenidos.

4.2. Calculadora Java

El primer caso de estudio se basa en un sistema desarrollado en Java en el cual uno de los componentes provee la funcionalidad de una calculadora, y al igual que el resto del sistema ha sido desarrollado dentro del proyecto original (*in-house*) y se lo denomina `JCalculator`. Este componente no presenta una interfaz gráfica para el usuario (GUI[1]) que actúe como cara visible externamente (*front-end*), sino que solo provee la funcionalidad principal relacionada a operaciones matemáticas, como se puede observar en la Figura 4.1(a). Dado que el sistema necesita evolucionar, algunos de sus

[1]del inglés *Graphical User Interface*

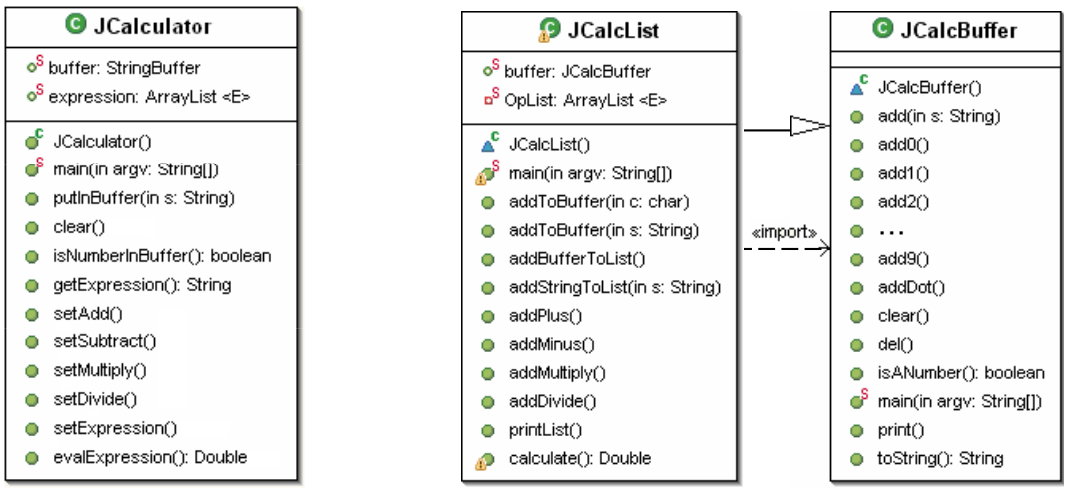
Componente	**Sitio Web**	**Versiones**
JCalc	*http://sourceforge.net*	0.1a; 0.1b; 0.0.3a; 0.0.4a; 0.1.5
NumberMonkey	*http://sourceforge.net/projects/NumberMonkey/*	3.0b
TerpCalc	*http://www.cs.umd.edu/∼atif/TerpOfficeWeb/TerpOffice/TerpCalc/*	4.0
GCalc	*http://gcalc.net, http://sourceforge.net*	3.0
JAC	*http://www.samnhum.net/, http://sourceforge.net*	1.1.232
JSciCalc	*http://jscicalc.sourceforge.net*	2-0.3
OpenCalculator	*http://sourceforge.net/projects/openCalculator/*	0.19
PocketCalc	*http://www.df.lth.se/∼mikaelb/, http://pocketCalc.softonic.com/pocketpc/*	1.1
SolCalc	*http://sourceforge.net*	1.1

Tabla 4.1: Calculadoras Java para evaluar compatibilidad con JCalculator

componentes pueden requerir un re-despliegue, y este es el caso de `JCalculator`, para el cual se ha considerado utilizar componentes externos como potenciales sustitutos. Del amplio espectro de calculadoras Java disponibles en el mercado actualmente, trece componentes fueron seleccionadas para evaluar su compatibilidad con `JCalculator`. La Tabla 4.1 presenta estos componentes candidatos: su nombre, el sitio web desde donde fueron descargados, y la versión (o versiones).

Para ilustrar claramente cada paso de las tres fases del Proceso de Evaluación, inicialmente se toma la primera versión del componente `JCalc` (es decir, la versión 0.1a), la cual incluye cinco ficheros java: tres de ellos correspondientes al *front-end* (GUI) del componente y los dos restantes que corresponden a sus funciones internas (*back-end*). La Figura 4.1(b) muestra las clases que conforman el *back-end* del componente, que representan la funcionalidad principal a ser evaluada en este caso de estudio. La evaluación de los componentes restantes será presentada en la Sección 4.3.

Con el propósito de lograr una decisión conclusiva sobre la compatibilidad entre `JCalculator` and `JCalc0.1a`, se debe iniciar la primera fase del Proceso de Evaluación, la cual involucra la creación de un TS de Comportamiento de Componente para `JCalculator`,



(a) JCalculator (b) JCalc

Figura 4.1: *Calculadora Java* – componentes software: Original (a) y de Reemplazo (b)

el cual será mas tarde utilizado para la evaluación semántica. A continuación se explica como procede esta fase.

4.2.1. TS de Comportamiento para JCalculator

Para construir el TS de Comportamiento de Componente para `JCalculator`, se deben cumplir los pasos que han sido automatizados en la herramienta *testooj* [Polo y otros, 2007] y que fueron explicados en la Sección 3.2 (pág. 62, del capítulo anterior). Es importante notar que algunos aspectos del diseño del TS de Comportamiento de Componente son responsabilidad del ingeniero de software, quien debe estar en conocimiento de los criterios de cobertura para componentes para los cuales se consideró una estrategia de implementación específica en nuestro Proceso de Evaluación. Por ello a medida que se avance con los pasos de esta fase, se explicarán las decisiones que tomaría un ingeniero de software para asegurar una adecuada cobertura para el TS.

Inicialmente se realizará la generación del TS de Comportamiento de Componente en formato JUnit con la intención de ejecutar sus casos de prueba contra el componente original (es decir `JCalculator` en este caso práctico), con el propósito de validar la cobertura alcanzada.

Uno de los pasos iniciales para construir el TS implica definir el *protocolo de uso* (en la

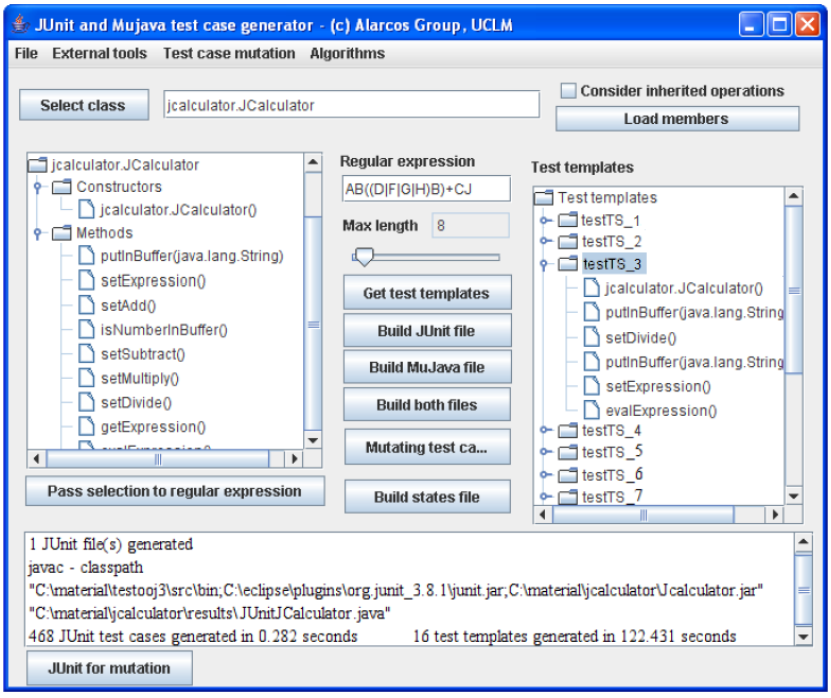


Figura 4.2: Protocolo de Uso para JCalculator

forma de una expresión regular), para lo cual un ingeniero de software debe utilizar una de las funcionalidades de la herramienta *testooj* que se puede observar en la Figura 4.2. Luego de cargar la clase `JCalculator`, sus constructores y métodos públicos se extraen de la clase y se muestran en el árbol que aparece a la izquierda de la ventana. Las operaciones de la clase se referencian con letras para el armado de la expresión regular y para `JCalculator` la expresión regular se ha definido como: $AB((D|F|G|H)B)^{+}CJ$, donde 'A' representa al constructor de la clase, 'B' representa al servicio `putInBuffer`, luego las operaciones matemáticas como opcionales, y con la posibilidad de iteración se encuentra también la segunda ocurrencia de `putInBuffer` (operaciones matemáticas de dos argumentos). Entonces la expresión regular anterior describe la siguiente expresión formada de servicios de `JCalculator`.

$$\texttt{JCalculator putInBuffer [(setAdd | setSubtract | setMultiply | setDivide) putInBuffer]}^{+}\ \texttt{setExpression evalExpression}$$

De la expresión regular anterior se crean plantillas de prueba (*test templates*), que describen las diferentes secuencias de invocaciones (eventos síncronos), y que son

generadas de acuerdo a la longitud que deben tener las sentencias que se derivan de una expresión regular (cantidad de símbolos del alfabeto incluidos en una sentencia). Dado que se necesita que el TS pueda cubrir el *criterio de dependencia del contexto*, de acuerdo a como se explicó en la Sección 3.2 (pág. 62, del capítulo anterior), analizamos a continuación cómo deberían ser las secuencias operacionales, iniciando con aquellas relacionadas a invocaciones a servicios (eventos síncronos).

La longitud mínima para las sentencias derivadas de la expresión regular anterior sería 6, lo cual podría resultar en cuatro sentencias que incluyen sólo un servicio matemático, como la secuencia de servicios de la plantilla `testTS_3` que se muestra en la parte derecha de la Figura 4.2 y que involucra al servicio `setDivide`. Estas cuatro plantillas con longitud de secuencia 6 permiten cubrir el *criterio de los alfabetos*, es decir que todos los símbolos del alfabeto involucrados en la expresión regular se incluyen en alguna de la sentencias derivadas.

Sin embargo, esta cobertura no es suficiente, y aun es necesario alcanzar el *criterio de los operadores* para así poder lograr satisfacer *criterio de dependencia del contexto*. Para ello se requiere en realidad una longitud mínima de 8, que produciría una iteración adicional para el operador '+' de la expresión regular, con lo cual se pueden generar otras dieciséis plantillas compuestas de dos servicios matemáticos y tres ocurrencias del servicio `putInBuffer`, como sucede por ejemplo en la plantilla número 6 que se observa a continuación:

```
JCalculator putInBuffer setAdd putInBuffer setSubtract putInBuffer
                    setExpression evalExpression
```

Por lo tanto, finalmente se generaron 20 plantillas de prueba que representan los dos grupos mencionados previamente, el primero de cuatro plantillas de un servicio matemático y el segundo de dieciséis plantillas de dos servicios matemáticos (dos iteraciones del operador '+').

Los siguientes pasos involucran establecer algunos otros aspectos tales como restricciones, excepciones y datos de prueba. La Figura 4.3 muestra los datos de prueba (*test values*) (`0, 1,3`) los cuales han sido asignados al único parámetro del servicio `putInBuffer`, y estos datos de prueba serán luego usados en pares de acuerdo al *protocolo de uso* – es decir, un valor antes y después de una invocación a un servicio matemático.

El comportamiento correcto de un componente puede requerir lanzar algunas excep-

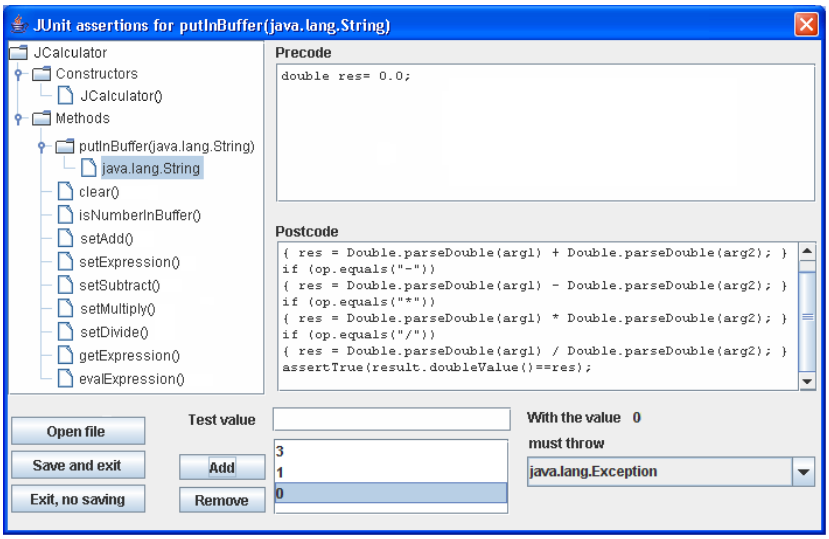


Figura 4.3: Restricciones, Excepciones y Datos de Prueba

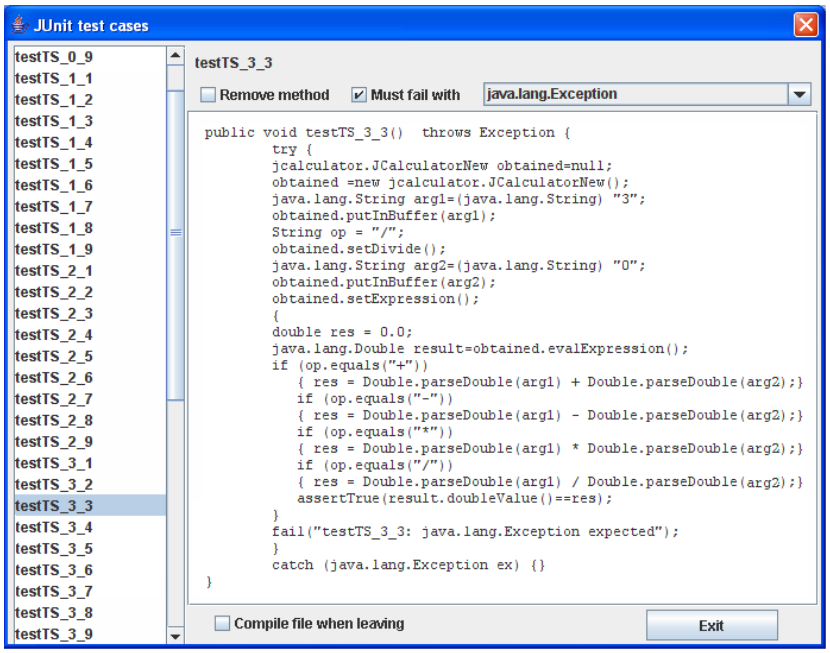


Figura 4.4: Casos de Prueba del TS para JCalculator en formato JUnit

ciones (eventos asíncronos), para lo cual se las extrae de la signatura de los servicios y se las muestra para especificar cuándo deberían ocurrir. En la parte inferir derecha de la de Figura 4.3 se muestra cómo se puede seleccionar una excepción para un servicio determinado y luego marcar que "*debe ser lanzado*" (*must be thrown*) con un dato de prueba cargado previamente. En el caso de `JCalculator` existe una excepción para el servicio `evalExpression` el cual debería ocurrir ante una división por cero, es decir cuando se invoque al servicio `setDivide`.

Dado que los casos de prueba en formato JUnit requieren que se incluya un oráculo, se necesita agregar algunas restricciones o aserciones (*assertions*) en las áreas de pre- y post-código como se observa en la Figura 4.3, que serán serializadas en ficheros. En particular para el oráculo se utiliza la clase *Assert* del framework de JUnit, la cual provee un conjunto de operaciones que ayudan a comprobar el estado de la clase en evaluación. Así por ejemplo en el post-código del servicio `evalExpression` se utilizó la operación *assertTrue* para comprobar si luego de evaluar la expresión matemática se alcanza el resultado esperado con los datos de prueba utilizados.

Luego de los pasos previos, se procede a combinar los datos de prueba con las 20 plantillas de prueba (secuencias operacionales) y los ficheros de restricciones (pre/post-código). Tales combinaciones se realizaron con la aplicación del algoritmo *all combinations* [Grindal y otros, 2005] provisto por la herramienta *testooj*, dado que así se logra cubrir todas las posibilidades de combinaciones que permiten obtener un TS de mayor envergadura, y que provea una confianza inicial mayor sobre la cobertura de diferentes aspectos, relacionados en este caso con las posibles interacciones de los clientes con el componente `JCalculator`.

De las 20 plantillas de prueba, 4 de ellas serán combinadas con pares de datos del conjunto de 3 datos de prueba, generando $4 * 3^2 = 36$ casos de prueba. Mientras que las 16 plantillas de prueba restantes serán combinadas con 3 datos de prueba, generando $16 * 3^3 = 432$ casos de prueba adicionales. Con lo cual finalmente se generaron 468 casos de prueba como métodos dentro de una clase denominada `JUnitJCalculator`. La Figura 4.4 lista los casos de prueba obtenidos en formato JUnit, y particularmente muestra uno de los casos de prueba, el método `testTS_3_3`, que surgió de la plantilla de prueba `testTS_3` mencionada previamente (ver la Figura 4.2), y que ejercita el servicio matemático `setDivide` con los datos de prueba 3 y 0 (cero) como argumentos (a través del servicio `putInBuffer`). Esto significa, que se producirá una división por cero, lo cual debería lanzar una *excepción*. Para ello, se ha insertado en el caso de prueba una

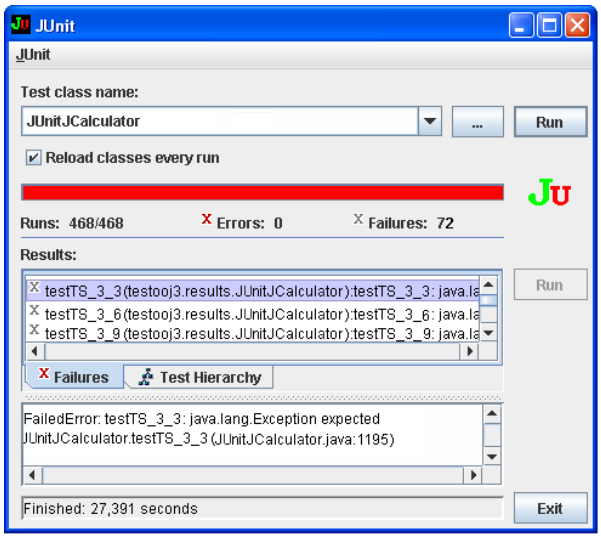


Figura 4.5: Resultados de la ejecución en JUnit del TS contra JCalculator

operación *Assert* adicional (*fail*) para capturar este tipo de excepción y así identificar apropiadamente la causa que lo origina.

De esta forma una *excepción* (evento asíncrono) para un determinado servicio queda inserta en el caso de prueba que invoca a ese servicio, y esto permite completar la cobertura del *criterio de eventos* en el TS. En la Sección 3.2 (pág. 62, del capítulo anterior), se explicó que inicialmente sólo se estaba logrando un cubrimiento parcial por medio del *protocolo de uso*, en el cual solamente se consideran las invocaciones a servicios (evento síncronos). Al cubrir adecuadamente el *criterio de eventos* también se refuerza el diseño del TS para la cobertura esperada del *criterio de dependencia del contexto*.

Luego de haber generado el TS, se procede a su validación mediante su ejecución contra el componente para el cual fue originalmente construido: `JCalculator`. Para ello se utiliza la función de *ejecución* de JUnit que es lanzada por la herramienta *testooj*, con la clase `JUnitJCalculator` (que representa el TS generado) para iterar a través de sus casos de prueba. La evaluación de los casos de prueba se realiza de acuerdo a las operaciones *Assert* que los mismos contienen, que actúa como oráculo y por lo tanto produce un resultado binario: éxito ó fallo. Los resultados se muestran en la Figura 4.5 donde 72 casos de prueba lanzaron las excepciones esperadas (ante una división por cero) y el resto obtuvieron resultados exitosos, validando así el TS generado para `JCalculator`.

Por lo tanto la clase java `JUnitJCalculator` representa el TS de Comportamiento de

```
public String testTS_3_3() {
  try {
      JCalculator obtained=null;
      obtained =new JCalculator();
      java.lang.String arg1=(java.lang.String) "3";
      obtained.putInBuffer(arg1);
      String op = "/";
      obtained.setDivide();
      java.lang.String arg2=(java.lang.String) "0";
      obtained.putInBuffer(arg2);
      obtained.setExpression();
      java.lang.Double result=obtained.evalExpression();
      return result.toString();
    }
      catch (Exception e) {
         return e.toString();
    }
}
```

Figura 4.6: Versión en formato MuJava de los Casos de Prueba para JCalculator

Componente para `JCalculator`, que era la meta a ser alcanzada en esta fase inicial del proceso, de manera que se puede continuar con el último paso de esta fase que implica derivar una versión del TS en formato MuJava, para ser usada en la tercera fase del proceso. Así se generó la clase java `MuJavaJCalculator` con mínimas variaciones: no fue necesario pre/post-código como tampoco un oráculo (dado que los métodos en formato MuJava retornan un valor de tipo String). La Figura 4.6 muestra el mismo método `testTS_3_3` en formato MuJava, donde se puede observar que las principales diferencias son el tipo de retorno y el hecho que todos los casos de prueba MuJava están preparados para levantar excepciones.

4.2.2. Validación Adicional del TS para JCalculator

Para medir la efectividad de los criterios seleccionados para el TS de Comportamiento de Componente como una estrategia para evaluar la compatibilidad de comportamiento, hemos realizado una validación adicional. El componente original `JCalculator` ha sido procesado por medio de una técnica de mutación de caja-blanca, lo cual resulta una estrategia común para medir la calidad de un TS, y en particular en este caso de estudio ha sido posible utilizar esta técnica dado que `JCalculator` es un componente *in-house* y por ello contamos con su código fuente.

Nivel	Operador	Descripción
Clase	JDC	Creación de un constructor por defecto
	JSD	Eliminación de modificador *static*
Métodos	AOIS	Inserción de atajo para operador aritmético
	AOIU	Inserción de operador aritmético unario básico
	AORB	Reemplazo de operador aritmético binario
	COI	Inserción de operador condicional
	LOI	Inserción de operador lógico
	ROR	Reemplazo de operador relacional

Tabla 4.2: Operadores de Mutación aplicados sobre JCalculator

Nivel	Operador	Método Afectado	Mutantes	Vivos	Muertos
Clase	JDC	– –	1	0	1
	JSD	– –	2	2	0
Métodos	AOIS	clear	2	2	0
		evalExpression	76	27	49
	AOIU	clear	1	0	1
		evalExpression	13	0	13
	AORB	evalExpression	32	0	32
	COI	isNumberInBuffer	1	0	1
		evalExpression	2	0	2
	LOI	clear	1	0	1
		evalExpression	8	0	8
	ROR	isNumberInBuffer	5	2	3
		Total	144	33	111

Tabla 4.3: Resultados de aplicar Operadores Mutantes sobre JCalculator

La Tabla 4.2 muestra el conjunto de operadores de mutación aplicados, los cuales están clasificados con dos niveles: *clase* y *métodos*, de acuerdo a [μJava Home Page, 2008; Offut, 1995]. Esta validación ha sido realizada por medio del soporte de la nueva versión de la herramienta MuJava la cual ha sido desarrollada como un *plug-in* para Eclipse, denominada MuClipse [MuClipse Home Page, 2008; Smith y Williams, 2007a,b], la cual está además basada en el framework de JUnit, aceptando un TS en formato JUnit para ejercitar los correspondientes mutantes.

Luego de aplicar los operadores de mutación, la herramienta generó 144 mutantes para `JCalculator`: 3 mutantes de clase y 141 mutantes de métodos, que se presentan en la Tabla 4.3. Luego se procedió a ejecutar la clase `JUnitJCalculator` contra los mutantes, es decir el TS de Comportamiento de Componente para `JCalculator` en formato JUnit.

El tiempo requerido para ejercitar los 468 casos de prueba contra los 144 mutantes es aproximadamente 5 horas. Los resultados indican que 33 mutantes permanecieron vivos y 111 mutantes resultaron muertos, lo cual implica un ratio de mutación de 77 %. Sin embargo, los mutantes que sobrevivieron incluyeron ciertos cambios en el código que los hizo equivalentes funcionalmente con respecto a `JCalculator`, lo cual significa que el TS desarrollado para `JCalculator` exhibió la efectividad requerida, exponiendo lo adecuado de los criterios de cobertura seleccionados para diseñar el TS de Comportamiento de Componente.

En la siguiente sección se explicará la segunda fase del proceso, la cual se aplica cuando un componente de reemplazo candidato debe ser integrado en el sistema.

4.2.3. Compatibilidad de Interfaces: JCalculator-JCalc01a

En la segunda fase del proceso se desarrolla un análisis de Compatibilidad de Interfaces que se realiza a nivel sintáctico entre el componente de reemplazo candidato `JCalc01a` y el componente original de referencia `JCalculator`, de acuerdo al esquema de cuatro niveles de equivalencia presentados en la Sección 3.3 (pág 73, del capítulo anterior).

Luego de cargar ambos componentes en la herramienta *testooj* y efectuar el análisis se presentan los resultados en un cuadro como se puede observar en la Figura 4.7. Dado que algunos de los servicios requeridos se heredan de una superclase en el componente `JCalc01a`, se ha optado por marcar la opción "*consider inherited operations*". El cuadro de resultados presenta en la columna de la izquierda los servicios del componente `JCalculator`, y hacia la derecha los servicios de `JCalc01a` con los cuales se identificó alguna correspondencia, que además se muestran ordenados por el nivel de equivalencia alcanzado, con el formato presentado en la Sección 3.3.2 (pág. 83, del capítulo anterior): [Caso, GradoEq, Signatura, Rx, Nx, Px, Ex].

En la Tabla 4.4 se presenta un resumen de los resultados, y a continuación se describen algunos de los aspectos interesantes de este análisis sintáctico de compatibilidad de interfaces:

- Se ha encontrado una correspondencia para todos los servicios de `JCalculator`, los cuales se muestran con una celda celeste en el cuadro de la Figura 4.7. Sin embargo, la excepción ha sido el servicio `evalExpression` que se muestra con una celda roja en la misma figura. Debería existir una correspondencia con el

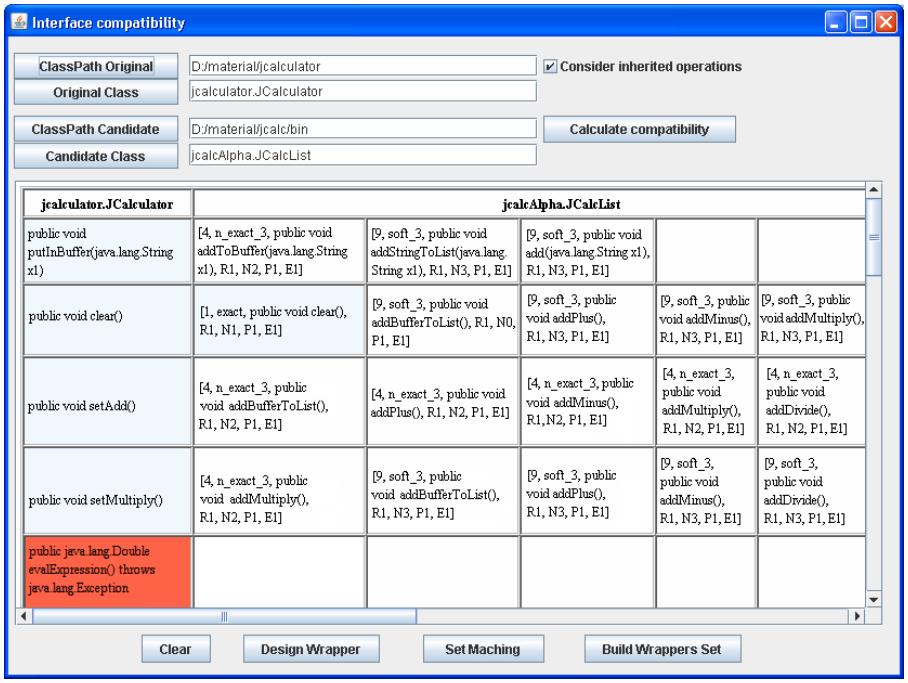


Figura 4.7: Compatibilidad de Interfaces entre los componentes JCalculator y JCalc

Exacto	Casi-Exacto	Moderado	Casi-Moderado	(Cantidad) Servicios de `JCalculator`	Mejor Valor
■ 1			(7)	getClass, toString, wait, etc.	■ 4
■ 1	■ 1	■ 20	(2)	notify, notifyAll	■ 4
■ 1		■ 21	(1)	clear	■ 4
	■ 1		(1)	isNumberInBuffer	■ 5
	■ 1	■ 2	(1)	putInBuffer	■ 5
	■ 1	■ 21	(2)	setMultiply, setDivide	■ 5
	■ 6	■ 16	(1)	setAdd	■ 5
		■ 1	(1)	getExpression	■ 6
		■ 22	(2)	setExpression, setSubtract	■ 6
		■ 1	(1)	evalExpression (*manual*)	■ 9
	Total servicios de `JCalculator`		19	Total Compatibilidad	92[§]

[§] Mejor Compatibilidad = 76

Tabla 4.4: Resumen de Compatibilidad de Interfaces para JCalculator-JCalc01a

servicio `calculate` de `JCalc01a`, siendo la condición E3 el aspecto incompatible: una lista de excepciones vacía en el servicio `calculate` mientras que en el servicio `evalExpression` existe una excepción para controlar la división por cero. Por lo tanto, para que el proceso no llegue a su fin, se requirió establecer manualmente la correspondencia entre estos dos servicios. Esta correspondencia ha sido clasificada como un caso especial de equivalencia *moderada* :: [25, soft_14, R1,N3,P1,E4] en la Tabla 4.4, con tipo de retorno (R1) idéntico, ignorando el nombre del servicio (E3), lista de parámetros idéntica (P1) y considerando un valor adicional para la condición de excepciones (E4) –ver Tablas 3.6 (pág 75), y 3.7 (pág 77) de la Sección 3.3 en el capítulo anterior. Esto por lo tanto genera un valor de equivalencia igual a 9 al sumar los valores de cada condición.

- Los métodos heredados de la clase *Object* fueron incluidos en la comparación, obteniéndose una correspondencia a nivel *exacto*. Por ejemplo el servicio `getClass` que obtuvo equivalencia *exacta* :: [1, exact, R1,N1,P1,E1], lo cual genera el valor de equivalencia 4. Los casos excepcionales son los servicios `notify` y `notifyAll` que tienen adicionalmente un relación cruzada con sus contrapartes en el componente `JCalc01a` con una equivalencia *casi-exacta* :: [4, n_exact_3, R1,N2,P1,E1]; y por otro lado obtuvieron una equivalencia *soft* con otros 20 servicios.

- Entre los métodos de la clase *Object*, que podrían ser no significativos para una comparación (como se discutió en la Sección 3.3, pág 73, del capítulo anterior), el servicio `toString` del componente `JCalc` resultó de importancia para producir una correspondencia con el servicio `getExpression` a nivel *soft* :: [9, soft_3, R1,N3,P1,E1], que de otra manera hubiera ocurrido que para un servicio significativo del componente original no se encontrara correspondencia, alcanzando así la situación explicada en el primer item.

- Un nivel alto de compatibilidad es importante para la siguiente fase del proceso como se explicará mas tarde. Este fue el caso con los primeros servicios mostrados en la Tabla 4.4, entre los cuales el servicio `clear` también obtuvo una equivalencia *exacta* – resaltado en la tabla de la Figura 4.7 con una celda en color celeste. Esto es un caso propicio dado que además este servicio obtuvo una equivalencia *soft* con 21 servicios de `JCalc01a` que podrían hacer mucho más complejo el procedimiento para la siguiente fase.

- Tres servicios obtuvieron una correspondencia a nivel *casi-exacto*, donde la equivalencia de substring (N2) permitió distinguir un caso de nivel mayor que *soft*. Entre estos servicios, `setDivide` y `setMultiply` presentan en particular un caso propicio similar al servicio `clear` dado que obtuvieron además una equivalencia *soft* con otros 21 servicios de `JCalc01a`. En la Figura 4.7 se puede observar el caso del servicio `setMultiply` que mapea al servicio `addMultiply` con equivalencia *casi-exacta* :: [4, n_exact_3, R1,N2,P1,E1].

- Tres servicios generaron las equivalencias más críticas, dado que obtuvieron correspondencia con más de un servicio de `JCalc01a`, y esto complejiza la decisión de compatibilidad. Tales servicios son `setAdd`, `setExpression` y `setSubtract`, que se pueden observar en la Tabla 4.4. En particular, el servicio `setAdd` logró mejorar el nivel de compatibilidad por encontrar equivalencias de substring (N2) como se pueden observar en la Figura 4.7.

Luego del análisis de los resultados y observando nuevamente el resumen presentado en la Tabla 4.4, se puede aplicar la Fórmula 3.1 (pág 78, de la Sección 3.3, en el capítulo anterior), para así calcular la *distancia sintáctica* entre `JCalculator` y `JCalc01a`, la cual sería: (92/76)-1=0,21. Luego en la Sección 4.3 se podrá observar un cuadro comparativo de distancias sintácticas con respecto al resto de los componentes candidatos con los cuales se efectuó el análisis de Compatibilidad de Interfaces.

El Mapeo de Compatibilidad obtenido en esta fase brinda la oportunidad de descubrir una potencial compatibilidad de componentes al proveer información para la siguiente fase que involucra la evaluación semántica de compatibilidad basada en pruebas.

4.2.4. Compatibilidad de Comportamiento: JCalculator-JCalc01a

Para iniciar el análisis de Compatibilidad de Comportamiento entre `JCalculator` y `JCalc01a` se necesita construir el conjunto de wrappers W de acuerdo al Mapeo de Compatibilidad generado en la segunda fase. Se ha considerado aquí tomar el nivel más alto de equivalencia posible para la construcción de los wrappers, omitiendo por lo tanto cualquier otra correspondencia existente de menor nivel para servicios con diferentes niveles de equivalencia.

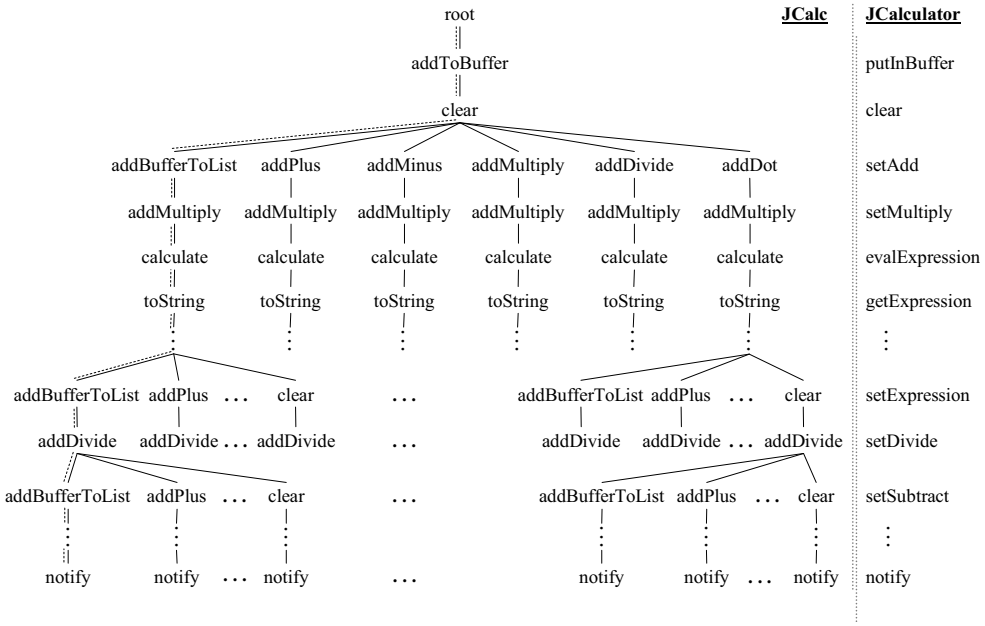


Figura 4.8: Árbol de generación de wrappers para JCalc01a

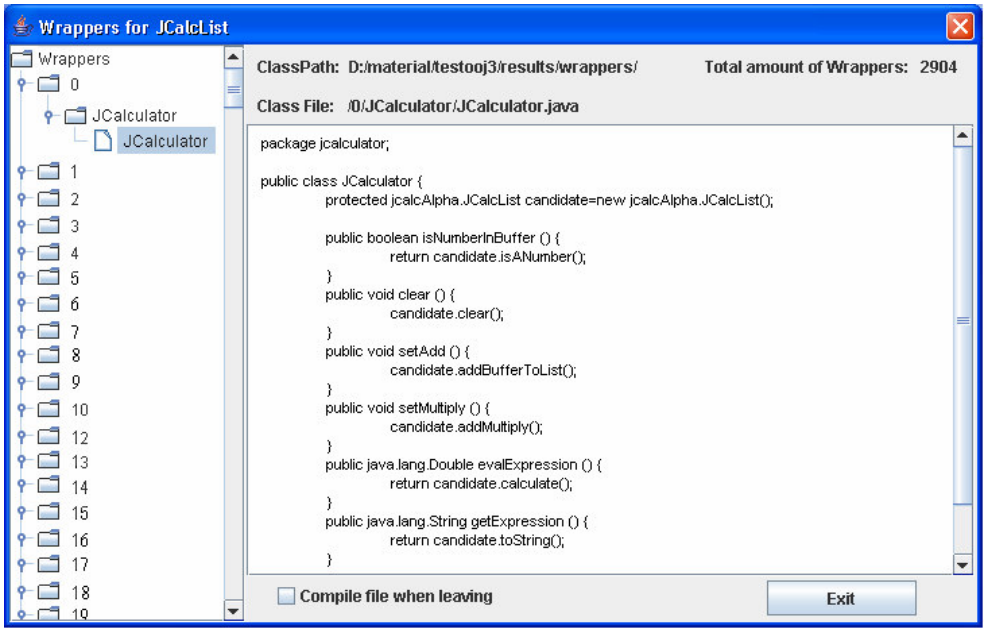


Figura 4.9: Wrappers de JCalculator para el componente JCalc01a

La Figura 4.8 muestra el árbol que se genera previo a la construcción de los wrappers, donde la cantidad de nodos en el nivel de las hojas asciende a 2904 (es decir el tamaño de W). Solamente tres servicios de `JCalculator` involucraron una correspondencia con más de un servicio de `JCalc01a` –como se puede observar en la Tabla 4.4 (pág. 106, de la sección anterior). La cantidad de correspondencias para tales servicios es entonces 6, 22 y 22 respectivamente, lo cual dentro de la Fórmula 3.2 (pág. 88, de la Sección 3.4, en el capítulo anterior) nos da el tamaño del conjunto W, es decir $6 * 22 * 22 = 2904$. La Figura 4.9 muestra cómo la herramienta *testooj* presenta la lista de adaptadores generados, donde en particular se puede observar el primer adaptador, el cual correponde al primer camino (desde la raíz al nodo hoja más a la izquierda) en el árbol de la Figura 4.8, marcado con líneas punteadas.

Luego de esto, el siguiente paso es ejecutar el TS de Comportamiento de Componente guardado en el fichero `MuJavaJCalculator` sobre cada adaptador del conjunto W para así evaluar la compatibilidad semántica. Para esto la herramienta *testooj* provee una facilidad particular denominada *executor*, la cual está basada en el framework de MuJava –ver la Figura 4.10. El executor toma el fichero de pruebas e itera a través del conjunto de wrappers. Luego de ello se puede observar el análisis de resultados para distinguir detalladamente los wrappers que han fallado en algún caso de prueba (marcados con `'X'`) cuando se comparan con el componente original `JCalculator` –ver la Figura 4.11. Los wrappers que han tenido un alto porcentaje de casos de prueba fallidos, se los considera como rechazados y por lo tanto se corresponden con mutantes muertos, dado que fueron generados por la aplicación de la técnica de *mutación de interfaces*.

La Tabla 4.5 muestra un resumen de los resultados donde solamente un wrapper pasó exitosamente todas las pruebas. Esto significa que sólo un wrapper puede sobrevivir (como caso de mutación), lo cual facilita la toma de decisiones en lo concerniente a no descartar el componente de reemplazo candidato (es decir `JCalc01a` en este caso práctico).

Aunque el rango de resultados exitosos es bastante alto, aún el porcentaje más alto por debajo del 100 % (es decir 77,77 %) tiene una diferencia distinguible, lo cual facilita reconocer que deben corresponder a versiones defectuosas del wrapper objetivo, es decir aquél que posee las correspondencias verdaderas que aquí se reconoce con el 100 %. En el caso del wrapper con 77,77 % de éxito, la única correspondencia errónea fue desde el servicio `setSubtract` hacia el servicio `del` de `JCalc01a`, en lugar de `addMinus` que representa la verdadera correspondencia.

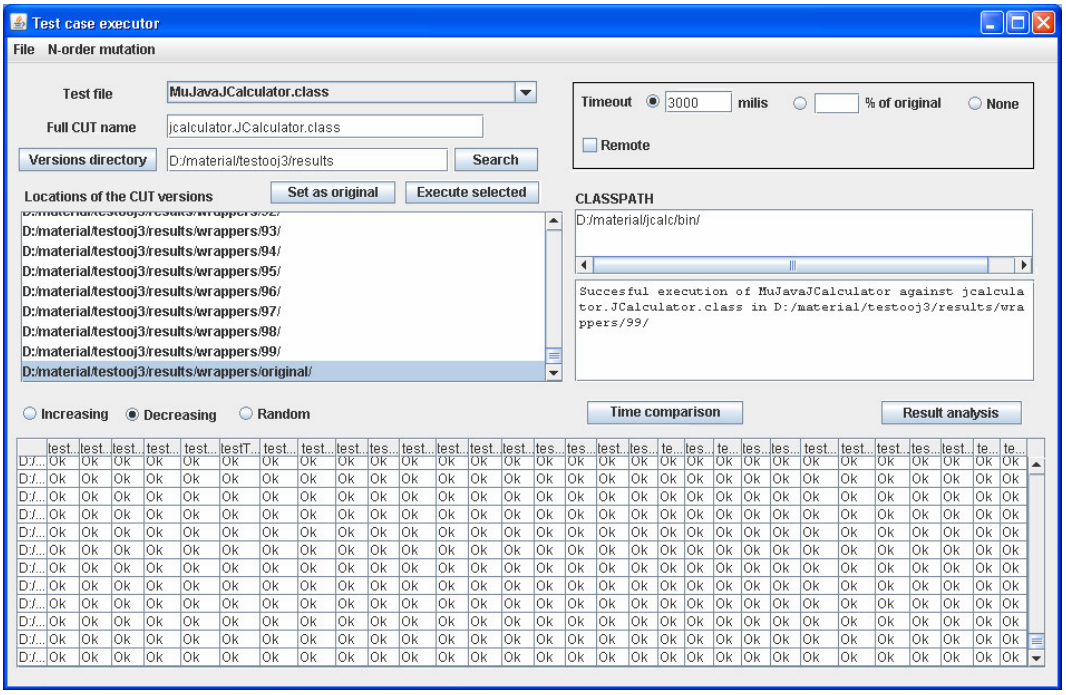


Figura 4.10: Execución del TS de JCalculator contra los wrappers para ejercitar el componente JCalc01a

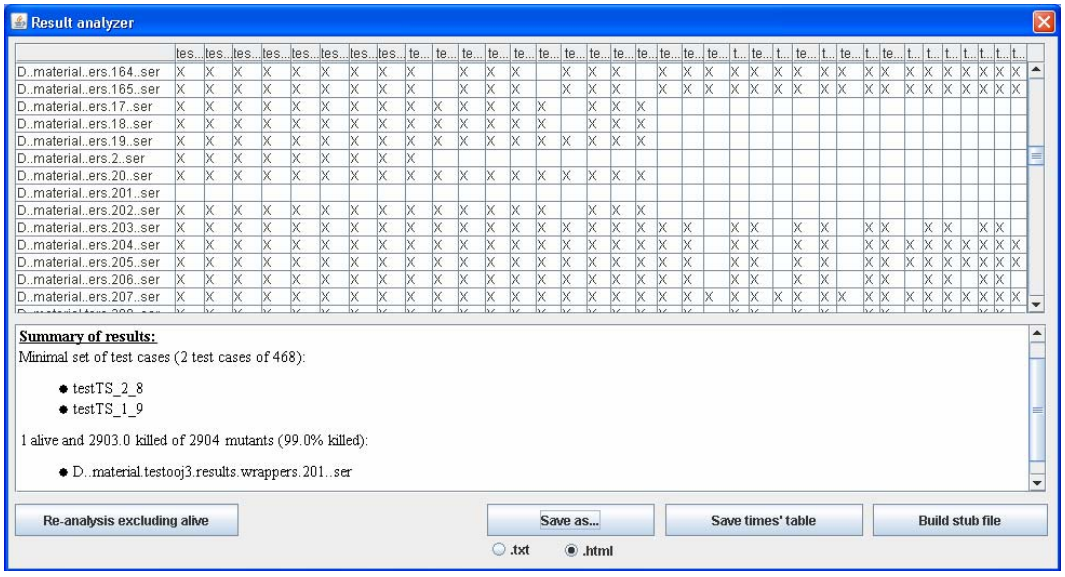


Figura 4.11: Resultados de la ejecución del TS de JCalculator sobre los wrappers de JCalc01a

Caso	% de Éxito	Wrappers
1	100	1
2	77,77	1
3	75	20
4	72,22	20
5	69,44	6
6	52,77	80
7	50	163
8	8,33	1290
9	0	1323
Total		2904

Tabla 4.5: Resumen de Resultados de ejecutar el TS de JCalculator sobre JCalc01a

El wrapper que sobrevivió no solo ayuda a descubrir la compatibilidad entre `JCalculator` y `JCal1ac0`, sino que además representa un artefacto potencial que un ingeniero de software puede utilizar para ajustar al componente candidato (`JCalc01a` en este caso práctico) para que pueda ser efectivamente integrado al sistema.

4.2.5. Reducción del Tamaño del Conjunto de Wrappers

El conjunto de wrappers podría adoptar un tamaño bastante grande de acuerdo a los casos de correspondencias identificados en la fase de Compatibilidad de Inferfaces. No obstante, la mayoría de tales casos ciertamente corresponderán a versiones defectuosas con respecto al wrapper objetivo – aquel que pueda describir las verdaderas correspondencias. Esto significa, que muchos wrappers en el conjunto, de hecho no califican como artefactos interesantes a ser considerados en una evaluación.

Por ejemplo, la cantidad de wrappers para el caso práctico desarrollado, podría crecer mucho más que cientos de millones cuando sólo se consideran equivalencias *moderadas*. En vez de ello, se ha conseguido reducir el conjunto de wrappers a sólo 2904 adaptadores, lo cual significa un tamaño muy alejado de ese gigantesto tamaño hipotéticamente analizado. Esto reduce el esfuerzo de las pruebas e incrementa el rendimiento para el proceso.

Las estrategias de reducción se aplican en la fase de Compatibilidad de Interfaces, con la intención de aumentar la cantidad de niveles de equivalencia más altos, es decir casos de equivalencia *exacta* o al menos *casi-exacta*. Por ello, el enfoque práctico que

Caso	% de Éxito	Wrappers
1	100	1
2	16,66	90
3	8,33	20
4	0	170
Total		281

Tabla 4.6: Resultados de ejercitar el Segundo grupo de Wrappers de JCalc01a

fue implementado involucra el análisis de los nombres de servicios para así encontrar equivalencias de substring –como se ha mencionado en la Sección 3.3 (pág. 73). En el caso práctico por ejemplo, para los servicios `setMultiply` y `setDivide` existe una equivalencia *casi-exacta* con `addMultiply` y `addDivide` respectivamente. Aún el servicio `setAdd` obtuvo una menor cantidad de correspondencias (6 en vez de 22), al utilizar esta estrategia.

Un segundo grupo de 281 wrappers ha sido construido en función de casos de *mutación de interfaces* no considerados en el grupo inicial de 2904 wrappers. Esto significa, que son menores los niveles de equivalencia aplicados en esta oportunidad. Los resultados luego de ejecutar el TS de Comportamiento de Components para `JCalculator` se pueden observar en la Tabla 4.6, donde un wrapper logró pasar exitosamente las pruebas, mientras que el resto obtuvo cero o un porcentaje muy bajo de éxito. Esto significa que existe otro wrapper que podría sobrevivir (como caso de mutación). Este segundo adaptador sobreviviente tiene la única diferencia de una correspondencia del servicio `putInBuffer` hacia el servicio `add(String s)` (ver la Figura 4.7), el cual de hecho también representa una correspondencia verdadera, aunque en realidad es un servicio heredado de una superclase en el componente `JCalc01a`.

Este segundo wrapper sobreviviente podría haber pasado sin ser reconocido en un proceso normal cuando sólo se consideran niveles de equivalencia altos. Sin embargo, la meta importante es ser capaces de reconocer apropiadamente una compatibilidad semántica, lo cual fue perfectamente logrado en el primer conjunto de wrappers, que estuve aún más cercano de encontrar sobrevivientes en la mayoría de sus miembros. Este segundo conjunto de 281 wrappers por el contrario, mas allá del sobreviviente estuvo demasiado lejos de encontrar sobrevivientes. Esto significa que el primer conjunto estaba en función de una base más sólida, lo cual expone la importancia del procedimiento de Compatibilidad de Interfaces.

4.3. Evaluación de 13 Componentes de Calculadora

Al inicio de la sección previa se mencionó que el caso de estudio desarrollado para la sustitución del componente de calculadora java `JCalculator` fue en realidad tratado con trece componentes. En la sección previa sólo se mostró una de tales evaluaciones, correspondiente al componente `JCalc01a`. En esta sección se presentará la evaluación del resto de los componentes para observar diferentes situaciones posibles cuando se manejan componentes desarrollados externamente (OTS).

Los trece componentes de calculadoras java han sido descargados de diferentes sitios web y usados para su comparación contra el componente `JCalculator`. La Tabla 4.7 muestra la cantidad de clases que comprenden cada componente (paquetes java) para así tener una percepción de su complejidad, junto con el resultado final de compatibilidad alcanzado (en forma reducida: aceptado o rechazado). Cada uno de los casos de evaluación será explicado con mayor detalle a continuación.

Componente	**Versiones**	**Contenido** (Clases)	**Código Fuente Disponible**	**Compatible**
JCalc	0.1a	5	si	✓
	0.1b	4	si	✓
	0.0.3a	6	si	X
	0.0.4a	6	si	X
	0.1.5	17	si	X
NumberMonkey	3.0b	5	si	✓
TerpCalc	4.0	32	si	✓
GCalc	3.0	>100	si	X
JAC	1.1.232	20	si	X
JSciCalc	2-0.3	>100	si	X
OpenCalculator	0.19	38	No	X
PocketCalc	1.1	3	si	X
SolCalc	1.1	3	si	X

Tabla 4.7: Calculadoras Java evaluadas contra JCalculator

4.3.1. Componentes Compatibles

La Tabla 4.8 presenta un resumen de la segunda y tercera fase del Proceso de Evaluación aplicado sobre los cuatro componentes compatibles, incluyendo la versión `0.1a` de `JCalc`, para así contrastar los resultados. Además, en la Tabla 4.9 se pueden

Componente	Clase Principal	Interfaces[$]		Comportamiento	
		Valor	Distancia	Wrappers	%Éxito
JCalc01a	JCalcList.class	92[§]	0,21	2904	100 %
JCalc01b	CalculatorList.class	92[§]	0,21	4608	100 %
NumberMonkey	CalcData.class	124[†]	0,63	2500	100 %
TerpCalc	TCBackendInterface.class	164[‡]	1,15	1458	100 %

[§] 1 correspondencia manual [†] 7 correspondencias manuales [‡] 8 correspondencias manuales
[$] Mejor Compatibilidad de Interfaces = 76

Tabla 4.8: Calculadoras Java compatibles a JCalculator

JCalc01a

Caso	% Éxito	W
1	100	1
2	77,77	1
3	75	20
4	72,22	20
5	69,44	6
6	52,77	80
7	50	163
9	8,33	1290
10	0	1323
Total		2904

JCalc01b

Caso	%Éxito	W
1	100	1
2	77,77	6
3	75	25
4	72,22	30
5	69,44	9
6	52,77	165
7	50	271
9	8,33	1779
10	0	2322
Total		4608

NumberMonkey

Caso	%Éxito	W
1	100	1
2	83,33	4
3	77,77	8
4	75	4
5	61,11	32
6	55,55	16
7	58,33	15
8	52,77	32
9	30,55	64
10	36,11	128
11	8,33	849
12	0	1347
Total		2500

TerpCalc

Caso	%Éxito	W
1	100	3
2	75	16
3	50	32
4	25	48
5	61,11	32
6	55,55	16
7	11,11	124
8	0	1235
Total		1458

Tabla 4.9: Resultados de ejecutar el TS de JCalculator sobre los componentes compatibles

observar los resultados de la ejecución del TS para `JCalculator` contra los cuatro componentes compatibles.

JCalc01b: Este componente es muy similar a la versión previa, aunque provee un servicio matemático adicional denominado `addNegative`. Aunque este produce una correspondencia adicional en la fase de Compatibilidad de Interfaces, el valor resultante se mantiene como en la primera versión de `JCalc` (ver Tabla 4.8), incluso ocurrió la misma situación de necesitarse una correspondencia manual para el servicio `evalExpression`. Sin embargo, las correspondencias adicionales afectaron la tercera fase, produciendo que el conjunto de wrappers tuviera un tamaño mayor: 4604 adaptadores. Como se puede ver en la Tabla 4.9, el conjunto de casos exitosos para `JCalc01b` es el mismo que para `JCalc01a`.

NumberMonkey: Para este componente la segunda fase reveló una importante situación de incompatibilidad sintáctica. La razón parece estar relacionada con una diferencia marcada en la distribución de funcionalidad entre los servicios del componente. Así el resultado inicial incluyó tres incompatibilidades para los servicios `putInBuffer`, `evalExpression` y `isNumberInBuffer`, para las cuales se requirió aplicar correspondencias en forma manual, para no finalizar el Proceso de Evaluación en esta instancia. Sin embargo, luego de efectuar una inspección profunda del resto de las correspondencias, se detectaron cuatro incompatibilidades adicionales, que involucran los servicios relacionados a las operaciones matemáticas (como `setAdd`). Por lo tanto, luego de solucionar esas incompatibilidades sintácticas, se pudo calcular la Compatibilidad de Interfaces, como se muestra en la Tabla 4.8, cuyo resultado es mayor que para los dos componentes previos, que implica una *distancia sintáctica* mayor entre `JCalculator` y `NumberMonkey`.

Sin embargo, para la tercera fase del proceso se generó un conjunto de wrappers de menor tamaño: 2500 adaptadores, como se observa en las Tablas 4.8 y 4.9, donde se descubrieron más casos de resultados exitosos, y un sólo wrapper pasó exitosamente el 100 % de los casos de prueba. Este último hecho permite tener una certeza de la compatibilidad entre los componentes que se comparan.

TerpCalc: En forma similar al componente `NumberMonkey`, el procedimiento de Compatibilidad de Interfaces descubrió una seria condición de incompatilidad para los servicios `isNumberInBuffer` y `evalExpression` –el último con un caso similar al de las primeras dos versiones del componente `JCalc`. Sin embargo, luego de un análisis profundo del resto de las correspondencias, se identificó la necesidad de agregar más correspondencias manualmente para otros seis servicios. Tales servicios están relacionados a las operaciones matemáticas junto con los servicios `putInBuffer` y `clear`. La Compatibilidad de Interfaces y por lo tanto la *distancia sintáctica* resultó con el valor mayor, como se observa en la Tabla 4.8, lo cual puede servir de indicador acerca de la complejidad para integrar el componente `TerpCalc` como un sustituto de `JCalculator`.

La tercera fase del proceso en este caso produjo el menor conjunto de wrappers: 1458, y también con el menor conjunto de casos exitosos, como se muestra en las Tablas 4.8 y 4.9. Además, del conjunto de wrappers se identificaron tres wrappers que pasaron exitosamente el 100 % de los casos de prueba. Tales wrappers involucran correpondencias verdaderas para todos los servicios, excepto para el servicio `clear`, el cual parece no

Componente	Incompatibilidad
JCalc003a	Completamente orientado al usuario. Sólo recibe datos de entrada a través de su *front-end* (GUI). No posee un *back-end* desacoplado
JCalc004a	Casi sin diferencia con JCalc003a
JCalc015	Muy diferente de las otras versiones. La entrada esperada implica una expresión matemática completa, en lugar de aceptar operandos y operadores para construir una expresión y luego efectuar la evaluación para obtener un resultado
GCalc	Difícil resolver las incompatibilidades sintácticas. Cambia la representación interna de una expresión matemática usando una estructura de árbol, debido a su propósito de graficar funciones matemáticas
JAC	Razón similar a GCalc, aunque no provee capacidad de graficar funciones matemáticas
JSciCalc	Razón similar a JAC
OpenCalculator	Razón similar a JAC y también a JCalc015
PocketCalc	Razón similar a JCalc003a
SolCalc	Razón similar a JCalc003a

Tabla 4.10: Calculadoras Java incompatibles con JCalculator

afectar la ejecución del resto de los servicios, por lo tanto la correspondencia manual agregada para este servicio no era realmente requerida.

4.3.2. Componentes Incompatibles

La Tabla 4.10 presenta un resumen de las razones de incompatibilidad descubiertas para cada uno del resto de los trece componentes usados para aplicar el Proceso de Evaluación para Sustitución de Componentes Software. Basicamente para los componentes `JCalc003a`, `JCalc004a`, `PocketCalc` y `SolCalc`, el problema que se encontró es que no se provee una interfaz *back-end*, sino solamente un *front-end* (GUI). Esto significa que estos componentes no fueron desarrollados para interoperar con otros componentes, sino sólo para trabajar en forma aislada. Para el resto de los componentes podría aplicarse alguna forma de adaptación para lograr la integración requerida. Sin embargo, dado que existe una diferencia masiva con respecto al componente `JCalculator`, el esfuerzo de adaptación podría resultar demasiado alto. Estos componentes representan calculadoras con una complejidad no esperada que no sería completamente explotada en el sistema objetivo.

4.3.3. Selección de Componentes

Luego de un escenario completo de los resultados obtenidos en las ejecuciones previas del Procedimiento de Evaluación para los diferentes componentes, se puede tomar una decisión final para seleccionar uno de los componentes como sustituto de `JCalculator`. Cuatro de los trece componentes resultaron con una compatibilidad evidente, y de este grupo las dos primeras versiones de `JCalc` implicaron un esfuerzo mínimo para agregar correspondencias manualmente. De estas dos versiones de `JCalc`, `JCalc01a` parece ser la consideración inicial como reemplazo para `JCalculator`. Sin embargo, el sistema donde se integrará el componente seleccionado, fue presentado en situación de evolución de acuerdo lo mencionado en la Sección 4.2 (pág. 95). Por lo tanto, una opción atractiva podría ser la selección del componente `TerpCalc`, dada su funcionalidad adicional relacionada con funciones matemáticas y las capacidades para graficarlas.

4.4. Proyecto JTopas

Se llevó a cabo otro experimento para observar la efectividad del proceso. En esta ocasión el conjunto de componentes fue descargado del repositorio público SIR (Software-artifact Infrastructure Repository) [Do y otros, 2005] *http://esquared.unl.edu/sir*, que tiene el propósito de ser usado como referencia para experimentos de pruebas.

Se seleccionó el paquete java `JTopas`, que provee un generador de tokens genérico y multi-propósito para texto reconocible (como código fuente, HTML, XML, y ASCII), y que puede ser integrado en un componente compilador. `JTopas` también está disponible en *http://jtopas.sourceforge.net*. Para este experimento se ha seleccionado la clase `PluginTokenizer` que representa el funcionalidad principal y hace uso del resto de las clases en el paquete. El proyecto completo de `JTopas` incluye 4 versiones, de donde la *version*0 (cero) fue considerada como el componente original de referencia, y las otras tres versiones se asumen por tanto como los componentes candidatos.

Se inició la primera fase del Proceso de Evaluación de Facilidad de Sustitución, para construir el TS de Comportamiento de Componente para la *version*0 de `PluginTokenizer`. Junto con el proyecto que se encuentra disponible en SIR, se provee adicionalmente un TS en formato JUnit, el cual fue utilizado como base para aprender

acerca del componente con la intención de desarrollar el TS correspondiente. Así se realizó uno de los pasos iniciales que conlleva describir el *protocolo de uso* (para representar secuencias operacionales) que permita lograr un TS adecuado para cubrir los criterios de cobertura requeridos – como se discutió en la Sección 3.2. Como resultado de este paso, se generaron tres plantillas de pruebas (*test templates*).

El TS descargado del proyecto además suministraba un conjunto de datos de pruebas *test data*, que consisten de 14 ficheros HTML a ser procesados para generar tokens. Estos datos de prueba fueron combinados con las tres plantillas de prueba para generar un TS compuesto de 42 casos de prueba (en formato JUnit) que fue guardado en un fichero denominado `JUnitPluginTokenizer`.

Luego de ello, el TS fue ejecutado contra la *version*0 de `PluginTokenizer`, que representa el componente original, para sí validar el TS. Dado que los resultados fueron exitosos entonces el siguiente paso fue la derivación de una versión del TS en formato MuJava que fue guardado en un fichero denominado `MuJavaPluginTokenizer`, para ser luego utilizado en la tercera fase del proceso.

Así a continuación se inició la segunda fase del proceso, que implica el análisis de Compatibilidad de Interfaces, cuyos resultados pueden observarse en la Figura 4.12, donde se muestra en particular con la *version*1 de `PluginTokenizer` como componente candidato. Como se puede ver los resultados involucran solamente equivalencias *exactas*. De hecho, la *version*1 tiene una interfaz más grande, lo cual se puede identificar perfectamente si se invirtiera el orden de evaluación. Es decir, si se asumiera la *version*1 como componente original, luego en la tabla de resultados se mostrarían celdas en color rojo para los servicios adicionales.

Luego de la *version*0, las restantes versiones en realidad mantienen la misma interfaz para la clase `PluginTokenizer`. Sin embargo, en la *version*3 se ha efectuado un cambio considerable, que básicamente involucra algunas optimizaciones a sus funciones internas.

Dado que se han reconocido sólo equivalencias *exactas*, entonces se requiere generar sólo un único wrapper. El único aspecto adicional podría involucrar aquellos servicios cuya lista de parámetros es mayor que uno. Algunos de ellos incluyen parámetros con tipos idénticos (o equivalentes) – como es el caso de `addBlockComment` con dos parámetros de tipo *String* (ver Figura 4.12). Tales parámetros podrían ubicarse en diferente orden (dentro de la lista de parámetros) para diferentes versiones. Sin embargo, dado que tales componentes representan realmente versiones subsiguientes (es decir, actualizaciones), se

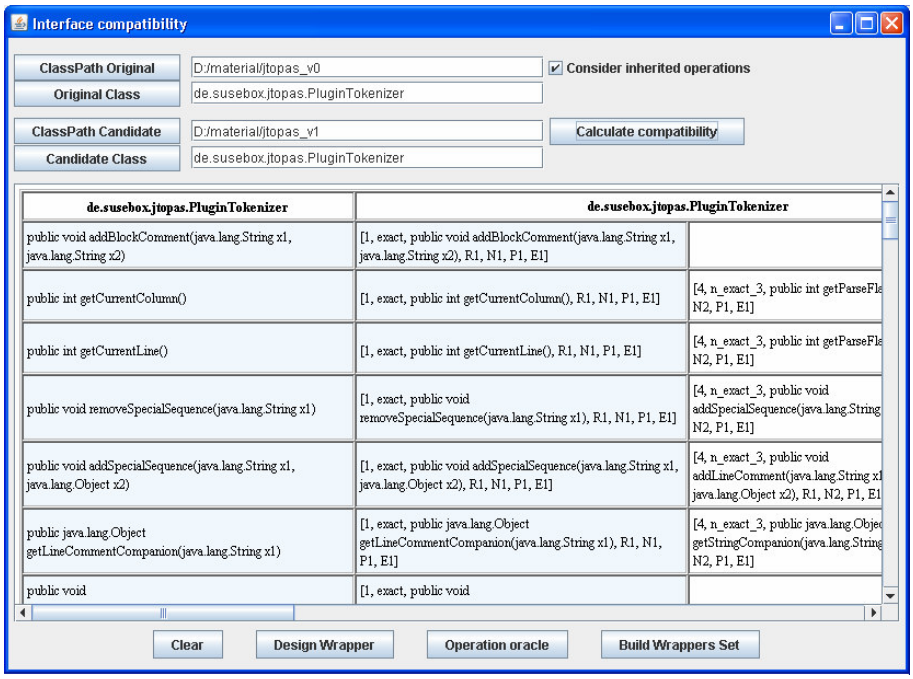


Figura 4.12: Interface Compatibility between *version*0 and 1 of `PluginTokenizer`

puede inicialmente asumir que no se efectuarán tales cambios en los servicios –en los cuales aparentemente no existe un cambio externo. Esto queda aún mas claro, cuando los principales cambios observados involucran el agregado de servicios extra en la interfaz (es decir, una extensión de la interfaz).

Por lo tanto se generó solamente un wrapper para cada una de las tres versiones restantes, de las cuales la ejecución del TS produjo resultados 100 % exitosos. Por lo tanto no se necesita la generación de algún adaptador adicional para tomar una decisión sobre la compatibilidad del conjunto de actualizaciones del proyecto `JTopas`.

En el caso que hipotéticamente el wrapper generado hubiera producido resultados fallidos, la siguiente opción estaría entre las combinaciones de parámetros para aquellos cuyos tipos son idénticos. Dado que cuatro servicios tienen dos parámetros similares y otros tres servicios tienen seis parámetros similares, el conjunto de wrappers podría aumentar a 3456 adaptadores, de acuerdo a la Fórmula 3.2 de tamaño del conjunto de wappers descripta en la Sección 4.2.4 (pág. 4.2.4, del capítulo anterior).

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Introducción

En este capítulo presentamos las principales contribuciones del trabajo de investigación realizado, y para ello se efectúa inicialmente un resumen del enfoque propuesto en esta tesis, en la sección siguiente. Luego en la Sección 5.3 se discuten los aspectos más importantes que permitieron alcanzar los objetivos planteados en la Sección 1.2 (pág. 10, del Capítulo 1). En la Sección 5.4 se indican las publicaciones en revistas y conferencias que permitieron obtener valiosas apreciaciones sobre los avances del trabajo de investigación. Finalmente en la Sección 5.5 se discuten las limitaciones del enfoque propuesto, y las líneas de trabajo futuro.

5.2. Resumen de la Investigación

El enfoque presentado en esta tesis se circunscribe particularmente a la etapa de mantenimiento del software, donde los sistemas basados en componentes requieren modificaciones por medio del reemplazo de ciertos componentes, ya sea con nuevas versiones (actualizaciones), nuevas entregas (*releases*), o bien con unidades de software completamente diferentes (es decir, componentes reutilizables de proveedores alternativos). El enfoque que se ha propuesto involucra un proceso específico, que hace uso de criterios de cobertura de pruebas para describir el comportamiento de los componentes, con el propósito de analizar la compatibilidad en componentes de reemplazo candidatos.

Por lo tanto, este Proceso de Evaluación para la Sustitución de componentes software, integra dos aspectos: evaluación de compatibilidad y actividades de pruebas del software, que provee por lo tanto una potencial reducción de esfuerzo para los ingenieros de software, mientras que suministra un soporte de fiabilidad.

La automatización de todo el proceso ha sido inicialmente soportada mediante la implementación de la herramienta *testooj*, la cual facilita la reducción de tiempo y esfuerzo, y permite reforzar el control sobre las condiciones de cada fase del proceso, para así alcanzar un enfoque de mayor rigor.

Dado que la herramienta *testooj* está particularmente dirigida a componentes Java, el siguiente paso involucra el desarrollo de funcionalidad adicional sobre una versión equivalente de la herramienta, que está enfocada en el framework .Net, en la cual actualmente se provee el soporte de ciertos aspectos del proceso.

De esta manera, el enfoque propuesto podría ser validado adicionalemente para un framework de componentes diferente, extendiendo así la aplicabilidad del Proceso de Evaluación y suministrando a los ingenieros de software un medio concreto para tratar con la Selección de Componentes en función de la necesidad de Sustitución de componentes software.

5.3. Consecusión de Objetivos

En la Sección 1.2 (pág. 10, del Capítulo 1) se presentaron los objetivos parciales que se plantearon para la consecución del *Objetivo* principal de esta tesis, que se presenta a continuación:

DEFINIR UN PROCESO DE EVALUACIÓN PARA LA SUSTITUCIÓN DE COMPONENTES SOFTWARE BASADO EN TÉCNICAS DE PRUEBA

A continuación analizamos la consecución de los objetivos parciales:

OP.1. Analizar el estado del arte en cuanto a métodos y técnicas para la sustitución y evaluación de componentes software.

En el Capítulo 2 se ha presentado un resumen de conceptos sobre Desarrollo de Software basado en Componentes (DSBC), en el cual se describen y comparan las

tecnologías más relevantes sobre *Modelos de Componentes*, que permiten diferenciar las formas de descripción e implementación de los componentes en cada tecnología, y los mecanismos de integración específicos mediante una *infraestructura de componentes* subyacente (Sección 2.1.1, pág. 23).

Por otra parte, se ha analizado el tratamiento que experimentan los componentes, a través de las principales actividades de una integración de componentes software: evaluación, adaptación, ensamblado, y actualización o modificación; las cuales además se consideran en función de las diferentes tecnologías y modelos de componentes. Asimismo se ha realizado una revisión de la literatura para identificar los enfoques más relevantes en relación con métodos de evaluación y propuestas de integración, los cuales son comparados para establecer las características de interés en un DSBC (Sección 2.1.2, pág. 26).

OP.2. Analizar el estado del arte en cuanto a técnicas de prueba de componentes software.

Dado que existen enfoques que aplican alguna técnica de prueba tradicional del software al contexto de DSBC, se ha presentado en el Capítulo 2 un resumen comparativo de conceptos y técnicas de Prueba Tradicional, tanto aquellas *basadas en especificación*, como aquellas *basadas en código*; identificando además su utilización de acuerdo a la típica división en fases de prueba del ciclo de vida tradicional: prueba de unidad, de integración, de sistema y de regresión (Sección 2.2.1, pág. 35).

Por otro lado se han presentado las características principales para Pruebas de Componentes, identificando las actividades de prueba desde las perspectivas de el *usuario* y el *proveedor* de los componentes; lo cual permite establecer estrategias y técnicas a aplicar en función de la disponibilidad de documentación y código fuente de los componentes reutilizables. Además se describen métricas para la facilidad de prueba de componentes y un conjunto de criterios de cobertura específicos para componentes, tanto a nivel de unidad como de integración de los mismos (Sección 2.2.2, pág. 42).

Además se han presentado las técnicas más relevantes para pruebas que tratan con la corrección y la fiabilidad del software basado en componentes. Algunas de tales técnicas, son aplicaciones directas de técnicas tradicionales, otras en cambio son el resultado de ajustes o variaciones de técnicas tradicionales, mientras que otras son propuestas nuevas desarrolladas particularmente desde las características propias que presenta el DSBC (Sección 2.2.3, pág. 48).

OP.3. Definir estrategias de comparación de interfaces de componentes, e implementar prototipos para llevar a cabo la experimentación.

En la Sección 3.3 (pág. 73, del Capítulo 3) se ha presentado el enfoque propuesto para efectuar la comparación de interfaces, para determinar la *distancia sintáctica* entre un componente de referencia (que requiere ser reemplazado) y otro componente candidato. Para ello se ha definido un esquema de *4 niveles de compatibilidad* para las operaciones declaradas en la interfaz de un componente: desde el más restrictivo (*exacto*) hasta el más débil (*casi-moderado*). Estos niveles de compatibilidad permiten clasificar los *grados de equivalencia* específicos (de los cuales se puede obtener un valor numérico), y que se calculan en función de los elementos que constituyen la signatura de las operaciones en una interfaz: tipo de resultado, nombre de la operación, tipos de los parámetros, y tipos de las excepciones.

Luego de efectuar el análisis comparativo de interfaz de los componentes involucrados, se genera un Mapeo de Compatibilidad que registra las correspondencias que ha obtenido cada operación del componente de referencia, con respecto a las operaciones del componente candidato. Cada correspondencia presenta un grado de equivalencia específico (y por lo tanto un nivel de compatibilidad), del cual se puede obtener un valor numérico, que permite finalmente efectuar un cálculo totalizado que determina la *distancia sintáctica* entre los componentes comparados.

OP.4. Definir estrategias de comparación de comportamiento de los componentes, e implementar prototipos para llevar a cabo la experimentación.

En el Capítulo 3 se presenta el enfoque de comparación de comportamiento de componentes, que establece por lo tanto un nivel semántico de evaluación. Este enfoque de evaluación se basa en analizar las funciones internas de los componentes que realizan transformación de datos, lo cual satisface la métrica de pruebas *facilidad de observación*, que se enfoca en el comportamiento operacional de un componente distinguiendo su salida como función de su entrada.

El enfoque está conceptualmente basado en la técnica de Pruebas Back-to-Back, la cual juzga la corrección de una nueva implementación en función de la generación de un conjunto de casos de prueba (TS, de *test suite*) para una implementación anterior (de referencia), y ejecutar ese TS contra ambas implementaciones para comparar los resultados [Vouk y otros, 1987; Shimeall y Leveson, 1991; Edwards, 2001; Veenendaal,

2006].

Así en nuestro enfoque, en cuanto surge la necesidad de que un componente deba ser reemplazado, se construye un TS específico cuyo objetivo es representar el comportamiento del componente, denominado TS de Comportamiento de Componente. Para el diseño de este TS se ha realizado una selección de criterios de cobertura de pruebas para componentes, utilizando una formalización mediante *expresiones regulares* como medio para una implementación efectiva dentro del enfoque (Sección 3.2, pág. 62).

Luego se necesita realizar una búsqueda de componentes candidatos, los cuales serán ejercitados contra el TS de Comportamiento de Componente, para identificar si existe un cierto grado conveniente de compatibilidad con respecto al componente que debe ser reemplazado. Para realizar esto, dado que el TS ha sido creado para el componente de referencia, y podrían existir diferencias de interfaz con un componente candidato, se utiliza el Mapeo de Compatibilidad de la comparación de interfaces de componentes (explicado para el objetivo anterior) para generar un conjunto de wrappers W (adaptadores) para un componente candidato, en base a la estrategia de *Mutación de Interfaz* [Gosh y Mathur, 2001; Delamaro y otros, 2001]. El objetivo final es encontrar un wrapper $w \in W$ que pueda reemplazar al componente de referencia, para permitir que sus componentes clientes puedan interactuar en forma segura con la interfaz del componente candidato (Sección 3.4, pág. 85).

OP.5. Definir un proceso de evaluación para la sustitución de componentes software.

Dado que se han definido dos enfoques de evaluación de componentes, a nivel sintáctico y semántico (como se explica en los dos objetivos anteriores), y esto ha requerido la definición de un enfoque de representación de componentes a utilizarse durante la evaluación; se han estructurado estos enfoques como fases de un proceso de evaluación, que permite la vinculación de las fases mediante los artefactos que se generan como salida de una fase y se requieren como entrada en otra subsiguiente.

En el Capítulo 3 se ha presentado este proceso de evaluación para la sustitución de componentes software, el cual inicia con la Fase 1 dedicada a la creación de un TS de Comportamiento de Componente para un componente que debe ser reemplazado (explicado en OP.5). Luego en la Fase 2 se realiza el análisis de Compatibilidad de Interfaces que determina la *distancia sintáctica* con respecto a un componente candidato y genera un

Mapeo de Compatibilidad (explicado en OP.4). En la Fase 3 se realiza el análisis de Compatibilidad de Comportamiento, utilizando como entrada el Mapeo de Compatibilidad de la fase previa (explicado en OP.5).

Cuando más de un componente candidato ha sido considerado para su evaluación, el proceso además permite efectuar *selección de componentes* en función de la información que se genera a través de los diversos análisis a los que se someten los componentes candidatos.

OP.6. Validar el enfoque mediante casos de estudio apropiados.

En el Capítulo 3 se ha ilustrado el proceso de evaluación (explicado en OP.5) mediante un ejemplo sencillo y ampliamente conocido sobre un sistema bancario, que ha permitido una primera aproximación a la aplicabilidad de nuestra propuesta. Luego en el Capítulo 4 se ha presentado un serie de experimentos, en función de componentes descargados de repositorios públicos, que permiten identificar que el proceso de evaluación propuesto no solamente tiene su aplicabilidad sobre el reemplazo de componentes que son versiones subsiguientes de un componente de referencia; sino que es también efectivo ante la decisión de utilizar componentes reutilizables de proveedores alternativos.

Uno de los experimentos se baso en un componente Java que provee la funcionalidad de una calculadora, y que debe ser reemplazada de un sistema desarrollado (*in-house*), para la cual se consideró la utilización de componentes externos como potenciales sustitutos. El experimento se realizó mediante la comparación con 13 componentes descargados de sitios web públicos (Sección 4.2, pág. 95).

Se llevó a cabo otro experimento para observar la efectividad del proceso. En esta ocasión el conjunto de componentes fue descargado del repositorio público SIR (Software-artifact Infrastructure Repository) [Do y otros, 2005] *http://esquared.unl.edu/sir*, que tiene el propósito de ser usado como referencia para experimentos de pruebas. Se seleccionó el paquete java `JTopas`, que provee un generador de tokens genérico y multi-propósito, y que puede ser integrado en un componente compilador. El proyecto completo de `JTopas` incluye 4 versiones, de donde la *version*0 (cero) fue considerada como el componente original de referencia, y las otras tres versiones se asumieron por tanto como los componentes candidatos (Sección 4.4, pág. 118).

OP.7. Desarrollar una herramienta que asista en la aplicación del proceso.

En el Capítulo 3 se explicó sobre la automatización de las tres fases del proceso de evaluación, mediante el desarrollo de la herramienta *testooj*. Luego en el Capítulo 4 se presentaron diferentes funcionalidades de la herramienta mediante su utilización en los casos de estudio desarrollados.

La herramienta *testooj* permite reforzar el control sobre las condiciones de cada fase del proceso, para así alcanzar un enfoque de mayor rigor y por lo tanto un soporte de fiabilidad.

Además, dado que un ingeniero de software deberá generalmente enfrentar las actividades de pruebas de componentes, ante el reemplazo de componentes; mediante esta herramienta se potencia la reducción de esfuerzo y tiempo dado que el proceso de evaluación está basado en la aplicación de criterios de cobertura de pruebas, y por lo tanto una parte de las pruebas requeridas durante la integración de un componente candidato, puede considerarse cubierto por el análisis de Compatibilidad de Comportamiento del proceso (explicado en OP.4), que está soportado por la herramienta *testooj*.

5.3.1. Alcances de la Propuesta

Del enfoque propuesto para la evaluar la facilidad de sustitución de componentes software, se puede identificar una serie de características tanto en función del beneficio provisto, como de algunas limitaciones que han sido señaladas por revisores expertos en el área, ante los avances presentados en diversas revistas y conferencias.

Beneficios **:** Algunas características a destacar de nuestra propuesta se detallan a continuación:

- La herramienta *testooj* permite reforzar el control sobre las condiciones de cada fase del proceso, para así alcanzar un enfoque de mayor rigor y por lo tanto un soporte de fiabilidad.

- Cuando más de un componente candidato ha sido considerado para su evaluación, el proceso además permite efectuar *selección de componentes* en función de la información que se genera a través de los diversos análisis a los que se someten los componentes candidatos.

- El proceso integra dos aspectos: evaluación y selección de componentes, y además actividades de prueba; con lo cual un ingeniero de software cuenta con una manera concreta de reducir el esfuerzo habitual que suponen las pruebas ante el reemplazo de un componente.

- La generación del TS de Comportamiento de Componente se realiza en base a la formalización con *expresiones regulares* (como se explica en la Sección 3.2, pág. 62), lo cual no supone un enfoque que pudiera complicar exageradamente la potencial adopción del proceso, por los desarrolladores en la industria.

- El proceso de evaluación propuesto no solamente tiene su aplicabilidad sobre el reemplazo de componentes que son versiones subsiguientes de un componente de referencia; sino que es también efectivo ante la decisión de utilizar componentes reutilizables de proveedores alternativos.

Limitaciones **:** Algunas de las posibles limitaciones que presenta nuestra propuesta se detallan a continuación:

- La herramienta *testooj* está particularmente dirigida a componentes Java, con lo cual tanto validación efectuada del proceso de evaluacion, como su aplicabilidad inmediata se encuentras limitadas al contexto en evaluación de componentes Java. Sería oportuno el desarrollo de una versión equivalente de la herramienta en otras tecnologías de componentes para ampliar la experimentación y su aplicabilidad.

- Actualmente la Fase 1 de Compatibilidad de Interfaces, efectúa el análisis de equivalencia para tipos de retorno y parámetros, solamente en función de tipos primitivos, por lo tanto se requiere una extensión a estructuras de tipos complejas (tales como arreglos, etc). En cambio para la generación del TS de Comportamiento de Componente que corresponde a la Fase 1, ya existe soporte para tipos complejos (Sección 3.3.1, pág. 79).

- Posiblemente el proceso que se ha propuesto, tenga su mayor aplicabilidad ante componentes que no presenten una envergadura demasiado compleja, lo cual generalmente se observa a través de sus interfaces. Según se puede observar en la Fase 2 de Compatibilidad de Comportamiento, existe un factor de escalabilidad ante la generación y ejecución de los wrappers (como casos de mutación de interfaz)

que afecta la eficiencia de esta fase. Aunque ya se han definido algunas heurísticas para reducir el conjunto de wrappers, sería conveniente ampliar el conjunto de heurísticas. Otro factor relacionado con la eficiencia es el tamaño del TS de Comportamiento de Componente, para el cual sería apropiado aplicar alguna estrategia de reducción del TS, considerando que no se vea afectada la eficacia y fiabilidad requerida del proceso (Sección 3.4, pág. 85).

5.4. Contraste de Resultados

Los resultados parciales obtenidos durante la investigación han sido publicados y presentados en diferentes foros, algunos de los cuales se presentan a continuación.

5.4.1. Publicaciones relacionadas con el Proyecto de Tesis

Revistas Internacionales

- *Testing-based Process for Component Substitutability.* **A. Flores**, M. Polo. En Revista *Software Testing, Verification and Reliability*. Wiley InterScience. John Wiley & Sons, Ltd. 2008.

  [*Aceptado con minor revisions*]. Índice de Impacto: 1,158 (2007)

  En este artículo, que se compone de 42 páginas, se describe todo el trabajo de investigación realizado y presentado en esta tesis doctoral, lo que incluye los últimas versiones de algoritmos diseñados e implementados en la herramienta *testooj*, y los últimos resultados experimentales.

- *Testing-Based Process for Evaluating Component Replaceability.* **A. Flores**, M. Polo. En Revista Electronic Notes in Theoretical Computer Science (ENTCS), vol. 236, pp. 101-115. Elsevier B.V. 2009.

  Este artículo surge de la invitación que se realiza a los autores para extender el paper presentado en la conferencia VODCA'08. En el mismo se describen los aspectos generales del enfoque presentado en esta tesis, aunque no se incluyen las últimas versiones de algoritmos de evaluación y los últimos resultados experimentales.

Conferencias Internacionales

- *Testing-based Assessment Process for Upgrading Component Systems.* **A. Flores**, M. Polo. En Actas de IEEE ICSM'08, 24^{th} International Conference on Software Maintenance. IEEE Computer Society Press. Beijing, China, Setiembre 2008.

  Ratio de Aceptación: 25,64 %

- *Testing-Based Process for Evaluating Component Replaceability.* **A. Flores**, M. Polo. En Actas de VODCA'08, 3^{rd} International Workshop on Views On Designing Complex Architectures. Bertinoro, Italia, Agosto 2008.
  "Seleccionado como mejor artículo y beneficiado con financiamiento (registración y asistencia) por Formal Methods Europe (FME, www.fmeurope.org)".

- *Testing-Based Component Assessment for Substitutability.* **A. Flores**, M. Polo. En Actas de ICEIS'08, International Conference on Enterprise Information Systems. INSTICC Press. pp. 386-939. Barcelona, España, Junio 2008.

- *Towards Software Component Substitutability through Black-Box Testing.* **A. Flores**, M. Polo. En Actas de STV'07, 5^{th} International Workshop on Systems Testing and Validation, en conjunto con ICCSEA'07, Fraunhofer IRB Verlag. pp. 111-120, Paris, Francia, Diciembre 2007.

Conferencias Iberoamericanas

- *Substitución de Componentes Software basado en Testing.* **A. Flores**, M. Polo. En Actas de WICC'07, 9^{no} Workshop Argentino de Investigadores en Ciencias de la Computación. RedUNCI. Trelew, Chubut, Argentina, Mayo 2007.

- *.Net Approach to Run-Time Component Integration.* **A. Flores**, I. García, M. Polo. En Actas de LA-WEB'05, 3^{rd} IEEE Latin American Web Congress. Regional conference by IW3C2, International World Wide Web Conference Committee <www.iw3c2.org>. IEEE Computer Society Press. Buenos Aires, Argentina. Octubre, 2005.

- *Dynamic Assembly & Integration on Component-based Systems.* **A. Flores**, M. Polo. En Actas de JIISIC'04, 4^{tas} Jornadas Iberoamericanas de Ingeniería del

Software e Ingeniería del Conocimiento. Madrid, España. pp. 349-360. Noviembre 2004.

5.4.2. Otras Publicaciones relacionadas

Capítulos de Libros

- *Applications Suitability on PvC Environments.* **A. Flores**, M. Polo. En Libro *Encyclopedia of Mobile Computing & Commerce*. Editores: David Taniar. Editorial: IGI Global (formerly Idea Group Inc.) ISBN: 978-1-59904-002. 2007.

Conferencias Internacionales

- *An Approach for Applications Suitability on Pervasive Environments.* **A. Flores**, M. Polo. En Actas de IWUC'06, 3^{rd} International Workshop on Ubiquitous Computing, durante ICEIS'06. INSTICC Press. Paphos, Chipre, Mayo 2006.

- *Dynamic Component Assessment on PvC Enviroments.* **A. Flores**, M. Polo. En Actas de ISCC'05, 10^{th} IEEE International Symposium on Computers and Communication. IEEE Computer Society Press. La Manga del Mar Menor, Cartagena, Murcia, España. Junio, 2005.

- *Towards Application Suitability for PvC Enviroments.* **A. Flores**, M. Polo. En Actas de MSVVEIS'05, 3^{rd} International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, durante ICEIS'05. INSTICC Press. ISBN 972-8865-22-8. pp. 58-62. Miami, Florida, USA. Mayo, 2005.

- *Towards Run-time Component Integration on Ubiquitous Systems.* M. Polo, **A. Flores**. En Actas de MSVVEIS'05, 3^{rd} International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, durante ICEIS'05. INSTICC Press. ISBN 972-8865-22-8. pp. 9-18. Miami, Florida, USA. Mayo, 2005.

- *Towards Context-aware Testing for Semantic Interoperability on PvC Environments.* **A. Flores**, J.C. Augusto, M. Polo, M. Varea. En Actas de SMC'04, 17^{th} International Conference on Systems, Man and Cybernetics, sesión especial:

Correctness and Reliability in Ubiquitous/Pervasive Computing. IEEE Computer Society Press. pp. 1136-1141. The Hague, Holanda. Octubre, 2004.

Conferencias Iberoamericanas

- *Considerations upon Interoperability on Pervasive Computing Environments.* **A. Flores**, M. Polo. En Actas de WICC'04, 6^{to} Workshop de Investigadores en Ciencias de la Computación. RedUNCI. Neuquen, Argentina, pp. 162-166. Mayo, 2004.

5.5. Trabajo Futuro

El enfoque que se ha propuesto en esta tesis, denominado Proceso de Evaluación basado en Pruebas para la Sustitución de Componentes Software, presenta una oportunidad de desarrollo en lo concerniente a líneas de trabajo futuro, tanto en función de brindar una solución a las limitaciones que fueran identificadas en la Sección 5.3, como en generar aplicaciones de la investigación desarrollada a nuevos campos relacionados con el DSBC.

Algunas de las líneas de trabajo futuro en particular en relación con nuevos campos de aplicación de la propuesta se exponen a continuación:

Aplicaciones Orientadas a Servicios

Como se expuso en la Sección 2.1.1 (pág. 23) del Capítulo 2, que describen las tecnologías más relevantes sobre *Modelos de Componentes*, los Servicios Web representan una forma de componentes reutilizables que pueden ser integrados a un sistema basado en componentes. Los Servicios Web se corresponden con aquellos componentes de los cuales no se dispone de código fuente para su evaluación, es decir, que se los considera de caja negra.

Dado que nuestro enfoque ha sido definido en función de la evaluación de componentes candidatos, de los cuales sólo se requiere conocer sus interfaces públicas para llevar a cabo el proceso de comparación de compatibilidades, entonces su aplicación al contexto de Servicios Web es inmediata.

De hecho ya se ha iniciado el trabajo sobre esta línea de investigación, en función de un proyecto argentino, que ha iniciado en enero de 2009, del cual el doctorando es miembro. El proyecto se denomina "Consolidación de la Ingeniería de Software Nacional con Miras a un Mercado de Software de Calidad Globalizado", Nº PAE 3/7279, con financiamiento de la Agencia Nacional de Promoción Científica y Tecnológica, Programa de Áreas Estratégicas: Competitividad de la Industria y modernización de sus métodos de producción. Su director es el Dr. Marcelo Campo, de la Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA) de Argentina; y el proyecto se compone de 13 universidades nacionales y 7 empresas, entre las que se encuentra la Universidad Nacional del Comahue, de la cual el doctorando es docente investigador.

Este proyecto se enfoca en la tecnología de Grid Computing que corresponde a un área de computación distribuida [Abbas, 2003; Berman y otros, 2003; Foster, 2003]. Los Grids prometen ofrecer recursos computacionales en cantidades casi ilimitadas, de manera fácil y transparente, simplemente enchufando programas a un Grid para aprovechar dichos recursos. En tal sentido, uno de los principales objetivos de un Grid es proveer una infraestructura computacional donde el software utiliza recursos, sin importar dónde se encuentran tales recursos y optimizando el uso de los mismos.

En particular el proyecto enfocará su investigación en Grids orientados a Servicios [Atkinson y otros, 2005], es decir, aquellos que se basan en el paradigma Service Oriented Computing (SOC) y las tecnologías de Servicios Web como forma de estandarizar y uniformar el acceso a recursos, tales como la arquitectura OGSA [Kishimoto y Treadwell, 2005].

El trabajo que el doctorando está realizando actualmente involucra por un lado la extensión de un prototipo (denominado easySOC [Crasso y otros, 2008]) que permite la identificación de Servicios Web, cuyo desarrollo inicial es un aporte del grupo de investigación de UNCPBA; y por otro la integración de ese prototipo con la herramienta *testooj* para aplicar el proceso de evaluación basado en pruebas al contexto de Servicios Web, y así evaluar los Servicios Web candidatos que fueran identificados por el prototipo easySOC.

Para llevar adelante esta investigación, el doctorando tiene a su cargo, la supervisión de una tesis de maestría y de varias tesis de grado (o proyectos de fin de carrera).

Entornos Pervasivos

Durante el primer período de doctorado, la investigación se encontraba inicialmente enfocada a la integración dinámica de componentes en los denominados Entornos de Computación Pervasiva (PvC[1]), es decir se consideraban estos entornos como una *infraestructura de componentes* subyacente para la integración y comunicación de componentes software.

De acuerdo a las consideraciones iniciales de Weiser [1991], un entorno PvC debería proveer la sensación de un entorno humano natural mejorado, en el cual los equipos de computación simplemente se desvanecen, dado que en general cuando las personas aprenden algo suficientemente bien, se genera un factor particular, y es que cesan de ser consientes de ello. Este hecho por lo tanto ocurre más por consecuencia de la psicología humana que de la propia tecnología. La relación de los usuarios con la computación cambia hacia una interacción humano-ordenador implícita, en la cual, en vez de pensar en términos de realizar tareas "en el ordenador" (tales como crear documentos, enviar e-mails, etc), en un entorno PvC los individuos se pueden comportar de forma natural, es decir, desplazarse a distintos sitios, usar objetos, observar y hablar unos con otros. El entorno es el encargado de facilitar estas acciones, y los individuos simplemente esperan la disponibilidad de los servicios que habitualmente utilizan, lo cual contribuye a la sensación de "continuidad" en sus actividades cotidianas [Cheng y otros, 2002].

Los usuarios deberían poder cambiar sus tareas computacionales entre diferentes contextos, y esto podría implicar el uso de una variedad de dispositivos móviles que ayudan a moverse dentro del entorno. Como resultado de ello, puede existir un cambio substancial en los recursos subyacentes para ejecutar las aplicaciones requeridas, tanto en espacio de memoria, capacidad de discos, y poder computacional, entre otros. Tales situaciones podrían hacer que un servicio o aplicación requerido no fuera apropiado en el nuevo contexto, con la probable necesidad de proveer un apropiado ajuste. Sin embargo, los usuarios no deberían percibir que el entorno circundante restringe de algún modo, sus actividades cotidianas. Debería existir un aprovisionamiento continuo de servicios y aplicaciones, y por lo tanto el entorno debería proveer un mecanismo para asegurar la *adecuación dinámica de aplicaciones* [Flores y Polo, 2006].

El objetivo de esta investigación estaba centrado en el ensamblaje dinámico (en tiempo de ejecución) de componentes para formar aplicaciones pervasivas, sobre las cuales se

[1] del inglés *Pervasive Computing Environments*

requieren continuas actualizaciones mediante el reemplazo de alguno de sus componentes constituyentes. Si bien, inicialmente este enfoque presenta similitudes con la propuesta de esta tesis, el cambio sustancial radica en la necesidad de realizar tales reemplazos en tiempo de ejecución, lo cual implica que las estrategias a aplicar deben tener fuertes consideraciones de eficiencia. Sin embargo, posiblemente se podría aplicar inicialmente el proceso de evaluación propuesto ante un entorno cerrado a un conjunto específico de aplicaciones y servicios, desde donde los componentes reutilizables que servirían de elementos constitutivos, serían completamente conocidos para un ingeniero de software a instancias de desarrollo, quien podría estimar los potenciales usos de tales componentes reutilizables, estableciendo servicios y aplicaciones en el entorno.

Para el caso de considerar la aplicación del proceso de evaluación bajo el requerimiento de evaluación de reemplazos en tiempo de ejecución, y dadas las consideraciones de eficiencia, se podría inicialmente aplicar la Fase 2 de Compatibilidad de Interfaces, la cual no demanda demasiados recursos, ni consume demasiado tiempo. Por otro lado se debería investigar sobre la evaluación de comportamiento, para diseñar un TS de Comportamiento de Componente que fuera mucho mas reducido y quizá con otros aspectos propios de las características de un entorno PVC, y por otro lado, y aun más importante, identificar nuevas estrategias para la ejecución de este TS sobre un componente candidato, para que esto no afecte la disponibilidad temporaria de aplicaciones en el entorno, y se mantenga por lo tanto transparente a los usuarios.

Bibliografía

Abbas, A. (2003). *Grid Computing: A Practical Guide to Technology and Applications*. Charles River Media.

Alexander, R. y Blackburn, M. (1999). «Component Assessment Using Specification-Based Analysis and Testing». *Informe técnico SPC-98095-CMC*, Software Productivity Consortium, Herndon, Virginia, EEUU.

Ammann, P. y Offutt, A (1994). «Using Formal Methods to derive Test Frames in Category-Partition Testing». En: *9th Annual Conference on Computer Assurance (COMPASS'94)*, pp. 69–80. IEEE Computer Society Press, Gaithersburg, MD, EEUU.

Andrews, A.; Ghosh, S. y Man-Choi, E. (2002). «A Model for Understanding Software Components». En: *International Conference on Software Maintenance (ICSM'02)*, pp. 359–369. IEEE Computer Socienty Press, Montreal, Canadá.

Andrews, J.; Briand, L. y Labiche, Y. (2005). «Is mutation an appropriate tool for testing experiments?» En: *27th International Conference on Software Engineering (ICSE'05)*, pp. 402–411. ACM Press, St. Louis, MO, EEUU.

Atkinson, C.; Gross, H-G y Barbier, F. (2003). *Component-Based Software Quality: Methods and Techniques*. volumen 2693 de *LNCS*, capítulo Component Integration through Built-In Contract Testing. Springer-Verlag.

Atkinson, M.; DeRoure, D.; Dunlop, A.; Fox, G.; Henderson, P.; Hey, T.; Paton, N.; Newhouse, S.; Parastatidis, S.; Trefethen, A.; Watson, P. y Webber, J. (2005). «Web Service Grids: An Evolutionary Approach». *Concurrency and Computation: Practice and Experience*, **17(2-4)**, pp. 377–389.

Avison, D.; Lau, F.; Myers, M. y Nielsen, P.A. (1999). «Action Research». *Communications of the ACM*, **42(1)**, pp. 94–97.

Bailey, E. (1997). *Maximum RPM*. Sams.

Ball, T.; Hoffman, D.; Ruskey, F.; Webber, R. y White, L. (2000). «State generation and automated class testing». *Software Testing, Verification and Reliability*, **10(3)**, pp. 149–170.

Baskerville, R.L. (1999). «Investigating Information Systems with Action Research». *Communications of the Association for Information Systems*, **2(19)**. Http://cis.gsu.edu/∼rbaskerv/CAIS_2_19/index.html.

Bass, L.; Clements, P. y Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley.

Becker, C. y Schiele, G. (2003). «Middleware and Application Adaptation Requirements and Their Support in Pervasive Computing». En: *23th ICDCSW'03*, pp. 98–103. IEEE Computer Society Press, Providence, Rhode Island, EEUU.

Becker, S.; Brogi, A.; Gorton, I.; Overhage, S.; Romanovsky, A. y Tivoli, M. (2006). «Towards an Engineering Approach to Component Adaptation». *Architecting Systems with Trustworthy Components*, **3938**, pp. 193–215.

Berman, F.; Fox, G. y Hey, A. (2003). *Grid Computing. Making the global infrastructure a Reality*. Wiley.

Bertolino, A. y Marchetti, E. (2005). *Software Engineering: The Development Process*. volumen 1, capítulo A Brief Essay on Software Testing. Wiley-IEEE Computer Society Press. 3^a edición.

Bertolino, A. y Polini, A. (2003). «A Framework for Component Deployment Testing». En: *25th International Conference on Software Engineering (ICSE'03)*, pp. 221–231. IEEE Computer Society Press.

Beydeda, S. y Gruhn, V. (2001). «An integrated testing technique for component-based software». En: *AICCSA ACS/IEEE International Conference on Computer Systems and Applications*, pp. 328–334. IEEE Computer Society Press, Beirut, Libanon.

Beyeda, S. y Ghrun, V. (2005). *Testing Commercial-off-the-Shelf Components and Systems*. capítulo Testing Component-based Systems Using FSMs. Springer-Verlag, Germany.

Binder, R.V. (2000). *Testing Object Oriented Systems - Models, Patterns and Tools*. Addison-Wesley.

Brada, P. y Valenta, L. (2006). «Practical Verification of Component Substitutability Using Subtype Relation». En: *32nd Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06)*, Dubrovnik, Croacia.

Briand, L.; Labiche, I. y Sówka, M. (2006). «Automated, Contract-based User Testing of Commercial-Off-The-Shelf Components». En: *28th International Conference on Software Engineering (ICSE'06)*, pp. 92–101. IEEE Computer Society Press, Shanghai, China.

Brown, A. y Wallnau, K. (1996). «Engineering of Component-Based Systems». En: *2nd International Conference on Engineering of Complex Computer Systems (ICECCS'96)*, pp. 414–422. IEEE Computer Society Press, Montreal, Canadá.

Budd, T.A. (1981). *Mutation Analysis: Ideas, Example, Problems and Prospects*. capítulo Computer Program Testing. North-Holand Publishing Company.

Cai, K.Y.; Chen, T.Y.; Li, Y.C.; Ning, W-Y y Yu, Y.T. (2005). «Adaptive Testing of Software Components». En: *SAC'05*, pp. 1463–1469. Santa Fe, New Mexico, EEUU.

CCM (2002). «CORBA Components - Version 3.0». *Informe técnico formal/02-06-65*, Object Management Group, Inc.. http://www.omg.org.

Cechich, A. y Piattini, M. (2007). «Early detection of COTS component functional suitability». *Information and Software Technology*, **49(2)**, pp. 108–121.

Cechich, A.; Piattini, M. y Vallecillo, A. (2003). *Component-based Software Quality: Methods and Techniques*. volumen 2693 de *LNCS*. Springer-Verlag.

Cheng, S.; Garlan, D.; Schmerl, B.; Sousa, J.; Spitznagel, B.; Steenkiste, P. y Hu, N. (2002). «Software Architecture-based Adaptation for Pervasive Systems». En: *ARCS'02*, volumen 2299 de *LNCS*, pp. 67–82. Karlsruhe, Alemania.

Choi, B.; Mathur, A. y Pattison, B. (1989). «PMothra: Scheduling mutants for execution on a hypercube». En: *3rd ACM SIGSOFT Symposium on Software Testing, Analysis, and Verification*, pp. 58–65. ACM Press, Key West, FL, EEUU.

Chow, T.S. (1978). «Testing Software Design Modeled by Finite-state Machines». *IEEE Transactions on Software Engineering*, **4(3)**, pp. 178–187.

COM (2008). «Component Object Model Technology». Disponible en http://www.microsoft.com/com/.

Councill, W.T. (1999). «Third-party Testing and the Quality of Software Components». *IEEE Computer*, pp. 55–57.

Crasso, M.; Zunino, A. y Campo, M. (2008). «Easy Web Service Discovery: A Query-by-Example Approach». *Science of Computer Programming*, **71(2)**, pp. 144–164. Elsevier Science.

Crnkovic, I.; Chaudron, M. y Larsson, S. (2006). «Component-based Development Process and Component Lifecycle». En: *International Conference on Software Engineering Advances (ICSEA'06)*, pp. 44–50. IEEE Computer Society Press, Tahití, Polinesia Franciasa.

Czerwonka, J. (2006). «Pairwise Testing in Real World». En: *24th PNSQC*, pp. 419–430. Portland, OR, EEUU.

DaSilva, A.; Martínez, J.; López, L. y Redondo, L. (2007). «Exhaustif®: una herramienta de inyección de fallos por software para sistemas empotrados distribuidos heterogéneos». *RPM-AEMES*, **4(1)**, pp. 51–61. ISSN: 1698-2029.

Debian Project (2007). «Debian Packaging Manual». http://www.debian.org/doc/.

Delamaro, M.; Maldonado, J. y Mathur, A. (2001). «Interface Mutation: An Approach for Integration Testing». *IEEE Transactions on Software Engineering*, **27(3)**, pp. 228–247.

DeMichil, L.; Yalçinalp, L. y Krishnan, S. (2001). «Enterprise JavaBeans Specification Version 2.0».

DeMillo, R.; Lipton, R. y Sayward, F. (1978). «Hints on test data selection: Help for the practicing programmer». *IEEE Computer*, **11**, pp. 34–43.

Do, H.; Elbaum, S.G. y Rothermel, G. (2005). «Supporting Controlled Experimentation with Testing Techniques: An infrastructure and its Potential Impact». *Empirical Software Engineering: An International Journal*, **10(4)**, pp. 405–435.

D'Souza, D. y Wilis, A. (1998). *Objetcts, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley.

Duncan, I. (1993). *Strong mutation testing strategies.* Tesis doctoral, Department of Computer Science, University of Durham,, U.K..

Edwards, S. H. (2001). «A Framework for Practical Automated Black-box Testing of Component-based Software». *Software Testing, Verification and Reliability*, **11**, pp. 97–111. http://www.interscience.wiley.com.

Estay, C. y Pastor, J. (2000a). «Improving Action Research in Information Systems with Project Management». En: *2000 Americas Conference on Information Systems*, pp. 1558–1561. Long Beach, CA, EEUU.

Estay, C. y Pastor, J. (2000b). «Towards a project structure for Action-Research in Information Systems». En: *10th Annual Business and Information Technology Conference (BIT'00)*, pp. 1558–1561. Manchester, Reino Unido.

Estay, C. y Pastor, J. (2000c). «Un Modelo de Madurez para Investigación-Acción en Sistemas de Información». En: *VI Symposium of Software Engineering and Data Base (JISBD'01)*, pp. 265–281. Almagro, España.

Fabbri, S.; Maldonado, J.; Masiero, P. y Delamaro, M. (1995). «Mutation Analysis applied to validate specifications based on Petri Nets». En: *8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols (FORTE'95)*, pp. 329–337. Montreal, Canadá.

Fioukov, A.; Eskenazi, E.; Hammer, D. y Chaudron, M. (2002). «Evaluation of Static Properties for Component-Based Architectures». En: *28th EUROMICRO'02*, pp. 33–39. IEEE Computer Society Press, Dortmund, Alemania.

Flores, A. y Polo, M. (2006). «An Approach for Application Suitability on Pervasive Enviroments». En: *3rd International Workshop on Ubiquitous Computing (IWUC'06), durante ICEIS'06*, pp. 71–78. INSTICC Press, Paphos, Chipre.

Foster, I. (2003). «The Grid: Computing without Bounds». *Scientific American*, **288(4)**, pp. 78–85.

Freedman, R. S. (1991). «Testability of Software Components». *IEEE Transactions on Software Engineering*, **17(6)**, pp. 553–564.

French, W.I. y Bell, C.H. (1996). *Organizational Development: Behavioral Science Interventions for Organization Improvement.* Prentice Hall, Inc..

Fujiwara, S.; Bochmann, G.; Khendek, F.; Amalou, M. y Ghedamsi, A. (1991). «Test selection based on finite state models». *IEEE Transactions on Software Engineering*, **17**.

Gamma, E.; Helm, R.; Johnson, R. y Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gao, J.Z. y Wu, Y. (2003). *Testing and Quality Assurance for Component Based Software.* Artech House.

Gosh, S. y Mathur, A. P. (2001). «Interface Mutation». *Software Testing, Verification and Reliability*, **11**, pp. 227–247. http://www.interscience.wiley.com.

Gosling, J.; Joy, B.; Steele, G. y Bracha, G. (2005). *Java™ Language Specification.* Sun Microsystems, Inc. Addison-Wesley, EEUU, 3ª edición. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html .

Grieskamp, W.; Gurevich, Y.; Schulte, W. y Veanes, M. (2001). «Testing with Abstract State Machines». En: *Workshop on Abstract State Machines (ASM'01), durante Formal Methods and Tools for Computer Science*, pp. 257–261. Las Palmas de Gran Canaria, Islas Canarias.

Grindal, M.; Offutt, A. y Andler, S.F. (2005). «Combination Testing Strategies: a survey». *Software Testing, Verification and Reliability*, **15(3)**, pp. 167–199. http://www.interscience.wiley.com.

Gross, H.G. (2005). *Component-Based Software Testing with UML*. Springer-Verlag, Alemania.

Harrold, M.J.; Liang, D. y Sinha, S. (1999). «An Approach to Analyzing and Testing Component-based Systems». En: *1st Workshop on Testing Distributed Component-Based Systems, durante ICSE'99*, Los Angeles, CA, EEUU.

Harrold, M.J. y Rothermel, G. (1994). «Performing data flow testing on classes». En: *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 154–163. ACM Press, Nueva York, EEUU.

Heineman, G. y Council, W. (2001). *Component-Based Software Engineering - Putting the Pieces Together*. Addison-Wesley.

Hong, H.; Lee, I.; Sokolsky, O. y Cha, S. (2001). «Automatic test generation from statecharts using model checking». En: *1st International Workshop on Formal Approaches to Testing of Software (FATES'01)*, pp. 15–30. BRICS: Basic Research in Computer Science, Aalborg, Dinamarca.

Hou, S-S.; Zhang, L.; Xie, T.; Mei, H. y Sun, J-S. (2007). «Applying Interface-Contract Mutation in Regression Testing of Component-Based Software». En: *Internacional Conference on Software Maintenance (ICSM'07)*, pp. 174–183. IEEE Computer Society Press.

Hughes, I. y Seymour-Rolls, K. (2000). «Participatory Action Research: Getting the Job Done». *Action Research E-Reports*, **4**. http://www.fhs.usyd.edu.au/arow/arer/004.htm.

IEEE 610 (1990). «Standard Glossary of Software Engineering Terminology». *ANSI/IEEE Standard 610-12-1990*, IEEE Press, Nueva York, EEUU.

J2EE (2008). «Java 2 Platform, Enterprise Edition, Sun Microsystems». Disponible en http://java.sun.com/j2ee/.

Jaffar-UrRehman, M.; Jabeen, F.; Bertolino, A. y Polini, A. (2007). «Testing Software Components for Integration: a Survey of Issues and Techniques». *Software Testing, Verification and Reliability*, **17(2)**, pp. 95–133. http://www.interscience.wiley.com.

JUnit Home Page (2008). «JUnit.org Resources for Test Driven Development». http://www.junit.org/home.

Kim, S.; Clark, J. y McDermid, J. (2001). «Investigating the effectiveness of object-oriented testing strategies using the mutation method». *Software Testing, Verification and Reliability*, **11(4)**, pp. 207–225. http://www.interscience.wiley.com.

Kirani, S (1994). *Specification and Verification of Object-Oriented Programs.* Tesis doctoral, Computer Science, University of Minnesota, Minneapolis, EEUU.

Kirani, S. H. y Tsai, W-T. (1994). «Method Sequence Specification and Verification of Classes». *Journal of Object-Oriented Programming*, **7(6)**, pp. 28–38.

Kishimoto, H. y Treadwell, J. (2005). «Defining the Grid: A Roadmap for OGSA^{TM} Standards». *Informe técnico GFD-I.053*, OGSA-WG, Open Grid Services Architecture Working Group. Version 1.0. Disponible en: http://www.ogf.org/documents/GFD.53.pdf.

Kock, N. y Lau, F. (2001). «Information Systems Action Research: Serving Two Demanding Masters». *Information Technology and People*, **14(1)**, pp. 6–11.

Kruchten, P. (1998). «Modeling Component Systems with the Unified Modeling Language». En: *1st International Workshop on Component Based Engineering (CBSE), durante ICSE'98*, Tokio, Japón.

Kung-Kiu, L. y Zheng, W. (2007). «Software Component Models». *IEEE Transactions on Software Engineering*, **33(10)**, pp. 709–724.

Lewin, K. (1947). «Frontiers in Group Dynamics». *Human Relations*, **1(1)**, pp. 5–41.

Ma, Y.S.; Offutt, J. y Kwon, Y. (2005). «MuJava: An automated class mutation system». *Software Testing, Verification and Reliability*, **15(2)**, pp. 97–133. http://www.interscience.wiley.com.

Malaiya, Y. (1995). «Antirandom Testing: Getting the most out of Black-box Testing». En: *International Symposium on Software Reliability Engineering (ISSRE'95)*, pp. 86–95. IEEE Computer Society Press, Toulouse, Francia.

Mann, S.; Borusan, A.; Ehrig, H.; Groβe-Rohde, M.; Mackenthun, R.; Sunbul, A. y Weber, H. (2000). «Towards a component concept for continuous software engineering». *Informe técnico*, Fraunhofer ISST, Berlín, Alemania.

Mariani, L.; Papagiannakis, S. y Pezzè (2007). «Compatibility and Regression Testing of COTS-component-based software». En: *29th International Conference on Software Engineering (ICSE'07)*, pp. 85–95. IEEE Computer Society Press, Minneapolis, EEUU.

Mariani, L.; Pezze, M. y Willmor, D. (2004). «Generation of Integration Tests for Self-Testing Components». En: *International Workshop of Testing Methodologies (ITM'04) durante FORTE'04*, LNCS 3236, pp. 337–350. Springer-Verlag, Toledo, España.

McTaggart, R. (1991). «Principles of Participatory Action Research». *Adult Education Quarterly*, **41(3)**.

Monson-Haefel, R. (2004). *Enterprise JavaBeans*. O'Reilly & Assoc., 4ª edición.

MuClipse Home Page (2008). «Eclipse Plugin for the MuJava mutation engine». http://muclipse.sourceforge.net.

μJava Home Page (2008). «Mutation system for Java programs». Http://www.cs.gmu.edu/∼offutt/mujava/.

Myers, G. J. (2004). *The Art of Software Testing*. John Wiley and Sons Inc., 2ª edición.

.NET (2009). «Microsoft: .NET Component Model». Disponible en http://www.microsoft.com/net.

Offut, A.J. (1995). «A Practical System for Mutation Testing: Help for the common Programmer». En: *12th International Conference on Testing Computer Software*, pp. 99–109. Washintong, DC, EEUU.

Offutt, J.; Liu, S.; Abdurazik, A. y Ammann, P. (2003). «Generating test data from state-based specifications». *Software Testing, Verification and Reliability*, **13(1)**, pp. 25–53.

Offutt, J.; Rothermel, G.; Untch, R.H. y Zapf, C. (1996). «An Experimental Determination of Sufficient Mutant Operators». *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **5(2)**, pp. 99–118.

Orso, A.; Do, H.; Rothermel, G.; Harrold, M.J. y Rosenblum, D. (2007). «Using Component Metadata to Regression Test Component-based Software». *Software Testing, Verification and Reliability*, **17**, pp. 61–94. http://www.interscience.wiley.com.

Orso, A.; Harrold, M. y Rosenblum, D. (2000). «Component Metadata for Software Engineering Tasks». En: *2nd EDO'00*, Springer-Verlag LNCS 1999, pp. 129–144. Davis, CA, EEUU.

Paulk, M. y Curtis, B. (1993). «Capability Maturity Model, Version 1.1.» *IEEE Software*, **10(4)**, pp. 18–27.

Peterson, M. (1995). *DCE: A Guide to Developing Portable Applications*. McGraw-Hill.

PMI (2000). «A Guide to the Project Management Body of Knowledge». *Informe técnico*, Project Management Institute Communication, EEUU.

Polini, A. y Bertolino, A. (2005). *Testing Commercial-off-the-Shelf Components and Systems*. capítulo A User-Oriented Framework for Component Deployment Testing. Springer-Verlag, Germany.

Polo, M.; Piattini, M. y Garcia Rodriguez, I. (2008). «Decreasing the cost of mutation testing with 2-order mutants». *Software Testing, Verification and Reliability*. http://www.interscience.wiley.com.

Polo, M.; Tendero, S. y Piattini, M. (2007). «Integrating Techniques and Tools for Testing Automation». *Software Testing, Verification and Reliability*, **16(1)**, pp. 1–37. http://www.interscience.wiley.com.

Pressman, R. (2000). *Software Engineering: A Practitioner's Approach*. McGraw Hill, 5ª edición.

Rapps, S. y Weyuker, E. (1982). «Data Flow Analysis Techniques for Test Data Selection». En: *6th International Conference on Software Engineering (ICSE'82)*, pp. 272–278. IEEE Computer Society Press, Tokio, Japón.

Riggs, R. (1998). «Java^{TM} Product Versioning Specification». *Informe técnico – Online*, SUN Microsystems Inc., Noviembre. Disponible en http://java.sun.com/j2se/1.5.0/docs/guide/versioning/spec/versioningTOC.html.

Rosenblum, D.S. (1997). «Adequate testing of component-based software». *Informe técnico UCI-ICS-97-34*, University of California, Irvine, CA, EEUU.

Rothermel, G. y Harrold, M.J. (1996). «Analyzing Regression Test Selection Techniques». *IEEE Transactions on Software Engineering*, **22(8)**, pp. 529–551.

Ruiz, F.; Polo, M. y Piattini, M. (2002). «Utilización de Investigación-Acción en la Definición de un Entorno para la Gestión del Proceso de Mantenimiento del Software». En: *1st Workshop en Métodos de Investigación y Fundamentos Filosóficos en Ingeniería del Software y Sistemas de Información (MIFISIS'02)*, .

Sabnani, K. y Dahbura, A. (1988). «Protocol test generation procedure». *Computer Networks and ISDN Systems*, **15**, pp. 285–297.

Seaman, C.B. (1999). «Qualitative Methods in Empirical Studies of Software Engineering». *IEEE Transactions on Software Engineering*, **25(4)**, pp. 557–572.

Shimeall, T. y Leveson, N. (1991). «An Empirical Comparison of Software Fault Tolerance and Fault Elimination». *IEEE Transactions on Sosftware Engineering*, **17(2)**, pp. 173–182.

Smith, B. y Williams, L. (2007a). «An Empirical Evaluation of the MuJava Mutation Operators». En: *Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART-MUTATION)*, pp. 193–202. Windsor, Reino Unido.

Smith, B. y Williams, L. (2007b). «On Guiding the Augmentation of an Automated Test Suite via Mutation Analysis». *Empirical Software Engineering*. Online only, DOI 10.1007/s10664-008-9083-7.

Stuckenholz, A. (2005). «Component Evolution and Versioning State of the Art». *ACM SIGSOFT Software Engineering Notes*, **30(1)**, pp. 7–20.

SWEBOK (2004). *Guide to the Software Engineering Body of Knowledge*. volumen ISO/IEC TR 19759:2005(E), capítulo 5, Knowledge Area Description of Software Testing. Software Engineering Coordinated Committee (IEEE Computer Society Press), Los Alamitos, CA, EEUU. http://www.swebok.org.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2ª edición.

The OSGi Alliance (2005). «OSGi Service Platform, Release 4». Disponible en http://www.osgi.org/.

Torgersson, J. y Dorling, A. (2002). «Assessing CBD - What's the Difference?» En: *28th EUROMICRO'02*, pp. 332–341. IEEE Computer Society Press, Dortmund, Alemania.

Tran, V.; Liu, D. y Hummel, B. (1997). «Component-based systems development: challenges and lessons learned». En: *8th International Workshop on Software Technology and Engineering Practice (STEP'97) (including CASE'97)*, pp. 452–462. IEEE Computer Society Press, Londres, Reino Unido.

UML 2.0 (2003). «Unified Modeling Language: Superstructure - Version 2.0». *Informe técnico*, Object Management Group, Inc.. http://www.omg.org.

Veenendaal, E. (2006). «Standard glossary of terms used in Software Testing». *Informe técnico Version 1.2*, Glossary Working Party, International Software Testing Qualification Board (ISTQB), Holanda.

Vincenzi, A.; Maldonado, J.; Delamaro, M.; Spoto, E. y Wong, E. (2003). *Component-based Software Quality: Methods and Techniques*. volumen 2693 de *LNCS*, capítulo Component-Based Software: An Overview of Testing. Springer-Verlag.

Vinoski, S. (2003). «Integration with Web Services». *IEEE Internet Comp.*, **7(6)**, pp. 75–77.

Vouk, M.; McAllister, D.; Eckhardt, D.; Caglayan, A. y Kelly, J. (1987). «Effectiveness of Back-to-Back Testing. En Experiments in Fault Tolerant Software Reliability». *Informe técnico NASA-NAG-I-667/NCSU-CSC-TR-87-08*, National Aeronautics and Space Administration (NASA). Langley Research Center, Hampton, VA, EEUU.

Wadsworth, Y. (1998). «What is participatory Action Research?» *Action Research International*, **2**.

Warboys, B.; Snowdon, B.; Greenwood, R.; Seet, W.; Robertson, I.; Morrison, R.; Balasubramaniam, D.; Kirby, G. y Mickan, K. (2005). «An Active-Architecture Approach to COTS Integration». *IEEE Software*, pp. 20–27.

Weide, B. (2001). «Modular Regression Testing: Connections to Component-Based Software». En: *4th International Workshop on Component Based Engineering (CBSE'01)*, .

Weiser, M. (1991). «The Computer for the 21st Century». *Scientific American*, **256(3)**, pp. 94–104.

Weyuker, E. J. (1998). «Testing component-based software: A cautionary tale». *IEEE Software*, **(15)**, pp. 54–59.

Whittaker, J. (2000). «What is Software Testing? And Why Is It So Hard?» *IEEE Software*, **17(1)**, pp. 70–79.

Wigley, A.; Sutton, M.; MacLeod, R.; Burbidge, R. y Wheelwright, S. (2003). *Microsoft .NET Compact Framework (Core Reference)*. Microsoft Press.

Wood-Harper, T. (1985). *Research Methods in Information Systems*. capítulo Research Methods in Information Systems: Using Action Research, pp. 161–191. North-Holland.

Wu, Ye; Chen, Mei-Hwa y Offutt, Jeff (2003). «UML-based Integration Testing for Component-based Software». En: *2nd International Conference on Cots-Based Software Systems (ICCBSS'03)*, pp. 251–260. LNCS 2580, Springer-Verlag, Ottawa, Canadá.

Wu, Ye; Pan, Dai y Chen, Mei-Hwa (2000). «Techniques of Maintaining Evolving Component-based Software». En: *16th International Conference on Software Maintenance (ICSM'00)*, p. 236. IEEE Computer Society Press, San Jose, CA, EEUU.

Wu, Ye; Pan, Dai y Chen, Mei-Hwa (2001). «Techniques for Testing Component-based Software». En: *7th International Conference on Engineering of Complex Computer Systems (ICECCS'01)*, pp. 222–232. IEEE Computer Society Press, Skovde, Suecia.

Yoon, H. y Choi, B. (2004). «Effective test case selection for component customization and its application to Enterprise JavaBeans». *Software Testing, Verification and Reliability*, **14(1)**, pp. 45–70. http://www.interscience.wiley.com.

Zakhour, S.; Hommel, S.; Royal, J.; Rabinovitch, I.; Risser, T. y Hoeber, M. (2006). *The Java^{TM} Tutorial: A Short Course on the Basics*. Sun Microsystems, Inc. Prentice Hall, EEUU, 4ª edición. http://java.sun.com/docs/books/tutorial/.

Zaremski, A. M. y Wing, J. (1997). «Specification Matching of Software Components». *ACM Transactions on Software Engineering and Methodology*, **6(4)**.