Facultad de Informática
Universidad Nacional de La Plata

Master en Ingeniería de Software

Tesis de Posgrado

"Tests de Regresión Automáticos creados por el Usuario Final sobre Ambientes Persistentes Orientados a Objetos"

Lic. Diego Cano
diego.cano@lifia.info.unlp.edu.ar

Directora: Dra. Silvia Gordillo

La Plata - Buenos Aires
Argentina
Octubre de 2015

Agradecimientos

Agradezco a Raúl Conti, Administrador General de la Tesorería General de la Provincia, por el permiso que me ha dado para citar en este trabajo los lineamientos generales del Sistema Cuenta Única del Tesoro, que se utiliza como ejemplo.

A Silvia Gordillo, por acceder por segunda vez a dirigirme un trabajo de tesina, siempre con una excelente predisposición.

A Javier Bazzocco y Daniel Casenave, por el aporte en aspectos de diseño e implementación que me han resultado de invalorable ayuda.

A Tere, por todo el apoyo y comprensión sobre el tiempo invertido en este trabajo.

ÍNDICE

1. INTRODUCCIÓN

En esta sección se describe el objetivo del presente trabajo y se brinda una breve introducción al contexto actual en relación al tema tratado. Asimismo se mencionan algunos problemas actuales en la utilización de los Tests de Regresión (en adelante TR) y algunos trabajos relacionados, siempre atento a su aplicación en proyectos reales de la industria de software.

Se brinda además una descripción de las metodologías de diseño, implementación y especificación utilizadas en este trabajo, como así también un resumen de la estructura del contenido del mismo.

1.1. Objetivo

Generar un modelo y la implementación de un prototipo como prueba de concepto que permita aumentar la automatización de Tests de Regresión, permitiendo su creación y configuración por parte del usuario final (juegos de datos, operaciones a testear, resultados esperados), reduciendo al mínimo o incluso evitando totalmente la intervención de los programadores y testers.

Tanto la selección de la tecnología a utilizar como la detección de los problemas a resolver están basadas proyectos actuales de desarrollo de software de la industria, tal como el Sistema Cuenta Única del Tesoro de la Tesorería General de la Provincia de Buenos Aires.

1.2. Contexto y Problemas Actuales

Los TR se ejecutan cada vez que se modifica la funcionalidad sobre la que se enfocan directamente, o cuando ésta podría haber sido alterada de forma indeseada por efectos laterales al cambiar la funcionalidad de otros módulos del sistema.

Estos tests muchas veces se realizan de forma manual y, en los casos en que se realizan de forma automática, tal automatización no siempre se concreta desde el primer momento, sino cuando su ejecución (de forma manual y reiterada) ya se encuentra lo suficientemente estable.

Los tests automatizados suelen ser programados por los testers o desarrolladores, que poseen conocimientos de programación en algún lenguaje. Esto requiere una inversión de esfuerzo por parte de estos desarrolladores, siendo que frecuentemente tienen una gran demanda de esfuerzo en la programación y desarrollo de la nueva funcionalidad del sistema.

1.3. Tests de Regresión y Tests de Unidad

Los TR comparten varios aspectos con los Tests de Unidad (en adelante TU).

TU es el test de una o varias unidades de software relacionadas, que se ejecutan bajo procedimientos y juegos de datos predefinidos, con el objetivo de probar su funcionamiento de forma aislada del resto de los módulos del sistema. Un TU puede definirse a cualquier nivel de la jerarquía de diseño, desde un módulo aislado hasta un programa completo [IEEE_UT].

Las similaridades entre ambos tipos de tests, permiten, por lo tanto, que varias de las soluciones pensadas para los TU, puedan ser reutilizadas con pocos cambios para los TR.

A continuación se listan los aspectos principales de los TU que son muy similares a los de los TR, por lo que resultan de interés y utilidad para el presente trabajo.

- Objetivo: como se mencionó, el objetivo de los TU es testear unidades funcionales de forma tal de asegurar cierto nivel de calidad en su funcionamiento de forma aislada. Posteriormente, en el Test de Integración, se agrupan estas unidades y se testea su interrelación.

- Automatización: en la mayoría de los casos a estos tests se los agrupa en "tests suites" (o "baterías de tests") y son ejecutados de forma automática. Esta ejecución, suele llevarse a cabo ante la puesta en producción de nuevos "releases" (versiones del sistema que incluyen nueva funcionalidad o modificaciones de la ya existente) de forma de minimizar la probabilidad de efectos laterales ante cambios del software.

- Independencia de Escenarios: es fundamental que cada TU se ejecute de forma totalmente independiente de los demás, y siempre (en cada ejecución) en un ambiente con la misma configuración, los mismos datos y bajo las mismas condiciones contextuales.asdfasdfa

1.4. Trabajos Relacionados

Existen varios frameworks para TU que se utilizan actualmente en la industria, por ejemplo JUnit, para Java, y SUnit, para Smalltalk. Estos frameworks, a su vez, pueden formar parte de otros más abarcativos desde el punto de vista de la Ingeniería de Software, de forma que incluyen a los TU como una de sus fases (por ejemplo, frameworks de Integración Contínua).

Por otro lado, la enorme mayoría de los sistemas de la industria del software se ejecutan sobre una Capa de Persistencia que almacena su información, la cual por lo general consta de una Base de Datos Relacional. A su vez, la Capa del Modelo de negocios de los sistemas actuales suele estar diseñada y programada mediante tecnologías Orientadas a Objetos, entre las cuales el lenguaje Java [JDK] tiene gran presencia. Es por esto que para comunicar estas dos capas (una relacional y la otra orientada a objetos) se utiliza un Mapeador O/R (objeto-relacional) el cual, como producto concreto, frecuentemente es Hibernate por ser uno de los concebidos para tecnología Java.

En este contexto, ante la necesidad de realizar TU en Java pero considerando a su vez la capa de persistencia, es que surge un nuevo framework como extensión a JUnit: DBUnit.

De esta forma, la estructura de muchos sistemas de la industria está constituida por una base de datos relacional en la capa de persistencia, Java en la capa del modelo, un mapeador Hibernate entre ambas y TU mediante JUnit o DBUnit.

En cuanto a la automatización de TR, en el mercado existen herramientas de tipo "Capture and Replay" (o "Record and Playback"), como Selenium, WebDriver y QALiber. Este tipo de herramientas permite grabar, programática o interactivamente, las acciones del usuario y luego reproducirlas cuantas veces se desee. Sin embargo, estas herramientas poseen varias desventajas: requieren algún tipo de conocimiento sobre desarrollo de software, usualmente incorporan actividades irrelevantes, y son muy sensibles al comportamiento, a la interfaz, a los datos y al contexto [INTI].

1.5. Metodología de Diseño, Desarrollo y Especificación

El diseño de los modelos de este trabajo se realiza dentro del Paradigma Orientado a Objetos, considerando las buenas prácticas de amplia aceptación en la industria, desde sus niveles básicos [Rebecca] hasta los más avanzados [Gamma].

Para la implementación de estos modelos se utiliza Java [JDK], por ser un lenguaje

cercano a este paradigma y de amplio uso industrial como así también en la Tesorería General de la Provincia.

La especificación tanto de los modelos de diseño como de implementación se realizan mediante UML [UML], por ser este lenguaje uno de los más utilizados en la industria.

1.6. Estructura de Contenidos

El presente trabajo se divide en nueve capítulos y dos anexos.

Luego de la introducción realizada en este primer capítulo, en el segundo se contextualiza a los Tests de Unidad y Regresión como parte del proceso general de QA dentro de la Ingeniería de Software. Se describen brevemente los conceptos básicos, los procedimientos elementales y se los ubica dentro de un modelo por capas, que es el utilizado en el Sistema Cuenta Única del Tesoro (SCUT) de la Tesorería General de la Provincia.

Sobre la base conceptual descripta, en ese mismo capítulo se tratan aspectos más prácticos y operativos de testing. En este sentido se describen frameworks de testing de amplio uso en la industria y su relación a las metodologías de persistencia. Es así que se describen las generalidades de los frameworks de testing JUnit y DBUnit y su uso en ambientes persistentes relacionales.

En el tercer capítulo se abandonan momentáneamente los aspectos técnicos para dar lugar a una muy breve introducción al Modelo Financiero Cuenta Única del Tesoro, como así también a los aspectos funcionales del sistema informático actual que le da soporte, y finalmente, se explica también la forma en que en el proyecto se organizan las prácticas de testing sobre tal sistema, que son las que conciernen a la temática del presente trabajo.

Los contenidos mencionados implican que se abarcan también las dimensiones de ingeniería de software y de recursos humanos, tanto en número como en diversidad de formación, tareas y objetivos. Con esto se intenta dar una idea general del contexto en el que se planea aplicar la solución propuesta en este trabajo.

El cuarto capítulo da inicio al desarrollo, propiamente dicho, de la solución propuesta en este trabajo. En este sentido se describen las generalidades del Esquema de Testing propuesto, tanto el uso básico desde el punto de vista del usuario final como algunos aspectos técnicos elementales de las aplicaciones que se podrán testear en el marco de este esquema. Como resultado, el lector tendrá los conceptos básicos para afrontar la profundización de los mismos en los capítulos siguientes.

El quinto capítulo se enfoca en la componente fundamental del Esquema de Testing introducido en el capítulo anterior: la Aplicación de Testing. Se describe el diseño de su modelo básico orientado a objetos, introduciendo previamente los lineamientos de diseño que se establecen como ejes de tal proceso. Este modelo básico constituye un modelo de muy amplia aplicación, incluso excediendo el alcance de este trabajo, y podrá ser extendido y configurado para contextos particulares.

El sexto capítulo describe la extensión y configuración del modelo básico de la Aplicación de Testing para su uso en el contexto particular del testeo de aplicaciones con arquitectura orientada a servicios (SOA). Previamente, se describen algunos lineamientos de diseño que se siguen en el desarrollo de las extensiones del modelo básico. Estos lineamientos abarcan tanto la extensión propuesta en este capítulo (orientación a servicios) como la extensión propuesta en el capítulo siguiente.

En el séptimo capítulo se trata la extensión del modelo básico de la aplicación de testing para poder ser persistido mediante las técnicas actuales más aceptadas en la industria. Esta persistencia, que usualmente se realiza de forma directa, se torna muy complicada en el caso

de este trabajo debido a la naturaleza de la aplicación en juego que, por su propia concepción, demanda un muy alto grado de flexibilidad. En este sentido, se da lugar a discusiones que confrontan a las técnicas de persistencia con las técnicas de reflection, y como resultado de tales análisis se propone un modelo extendido que logra satisfacer ambos aspectos.

El octavo capítulo brinda una descripción detallada de los aspectos tecnológicos y de uso de las aplicaciones que conforman el Esquema de Testing,. Así, se desarrolla una explicación detallada de aspectos técnicos de arquitectura, implementación y uso de la Aplicación de Testing y Aplicación Objetivo por parte de usuarios finales mediante las interfaces gráficas construidas a tal fin.

En el noveno capítulo se desarrollan algunas conclusiones sobre todo lo expuesto como así también el delineamiento de algunas propuestas de trabajos futuros.

Por último se brindan dos anexos: uno de ellos con ejemplos de bajo nivel de detalle (código Java) sobre el uso standard de DBUnit y otro con instrucciones de instalación del software entregado.

2. INGENIERÍA DE SOFTWARE, CALIDAD Y FRAMEWORKS ACTUALES

En este capítulo se ubica a la Gestión de la Calidad dentro de la Ingeniería de Software, y como parte de sus prácticas se describen algunos conceptos básicos de testing. Posteriormente se describen las generalidades de algunos frameworks actuales para realizar tests de unidad: JUnit y DBUnit. Asimismo, para comprender la evolución del primer framework hacia el segundo, se desarrollan algunos conceptos básicos de arquitecturas con persistencia.

2.1. Ingeniería de Software y QA

En esta sección se describe la ubicación de QA dentro del framework de procesos establecidos por el Project Management Institute. Si bien estos procesos no son propios de la informática sino que abarcan varias disciplinas de la ingeniería, se describe su relación a la ingeniería de software.

2.1.1. Ubicación de QA en la Ingeniería de Software

El Project Management Institute [PMI] define un conjunto de procesos y prácticas que son comunes a la Gestión de Proyectos en general, por lo cual se aplican en diversas áreas, una de las cuales es la Ingeniería de Software.

Estas prácticas y procesos han sido establecidos con base en el análisis de una gran cantidad de proyectos reales, se han sistematizado en un Framework y éste ha sido documentado en el Project Management Body of Knowledge (PMBOK).

El Framework tiene como punto de partida las necesidades y expectativas de los Stakeholders. En este sentido, para dar cobertura a las mismas de forma ordenada y sistemática, define nueve Áreas de Conocimiento, cuatro de las cuales son considerados Procesos Centrales (core), entre los que se encuentra el área de Quality Management.

El punto crucial del Framework es el Proceso de Integración (Project Integration Management) de todos los demás procesos, y constituye una suerte de bus a partir del cual los Project Managers gestionan todo el proyecto.

Durante la ejecución del Framework se utilizan herramientas y técnicas; en este trabajo nos centraremos en herramientas y técnicas utilizadas en el área de Gestión de la Calidad en el contexto de proyectos de Ingeniería de Software.

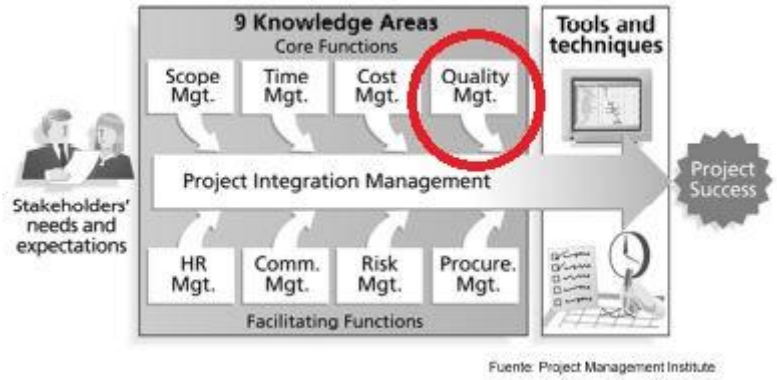


Figura 2.1 – QA en la Ingeniería de Software

2.1.2. Tipos de Tests

La Gestión de la Calidad, tanto en proyectos en general como en la Ingeniería de Software, abarca una gran cantidad de prácticas y técnicas. En esta sección describiremos brevemente una de ellas: los tests.

En Ingeniería de Software existen varios tipos de tests que pueden realizarse a un sistema, cada uno con un fin específico, algunos de ellos se describen brevemente a continuación:

- Tests de Unidad: se utilizan para verificar el funcionamiento de los módulos o unidades de un sistema, como por ejemplo una clase (OO) o un conjunto de ellas fuertemente acopladas. Esto último significa que son de bajo nivel, ejecutados y entendidos sólo por los desarrolladores del sistema. [IEEE]

- Tests de Integración: Ayudan a probar la interacción entre diferentes módulos o unidades de la aplicación que ya han sido probadas mediante tests de unidad. [IEEE]

- Test de Stress: se trata de un test no-funcional, y su objetivo es evaluar el rendimiento del sistema en situaciones de gran carga, demanda y exigencia.

- Tests de Regresión: constituyen el re-testeo de un sistema o componente para verificar que modificaciones realizadas al sistema no han causado efectos no deseados y que el sistema o componente aún cumple con los requerimientos especificados. [IEEE]

- Tests de Aceptación de Usuario (UAT, User Acceptance Test): son tests funcionales que se ejecutan sobre una versión final del sistema o módulo como un todo, con el objetivo de ser evaluado y aprobado por los usuarios finales, lo cual persigue metas contractuales como el cierre de una etapa o proyecto.

Existen muchos otros tipos de tests que no viene al caso mencionar aquí, entre los cuales se encuentran: tests exploratorios, de seguridad, de usabilidad, de escalabilidad, de mantenibilidad, de instalabilidad, de portabilidad.

En este trabajo nos centraremos en los Tests de Regresión y Unidad en sistemas Orientados a Objetos en donde, en particular, existe una Capa de Persistencia.

2.2. Tests de Regresión Automatizados – Herramientas actuales

En esta sección se describen algunas de las típicas herramientas actuales utilizadas en la industria para realizar tests de regresión automatizados, tanto programados mediante APIs como de tipo “Capture and Replay” (o “Record and Playback”). En concreto, se describen las herramientas Selenium IDE, WebDriver y QALiber. Este tipo de herramientas permite grabar las acciones del usuario, programática o interactivamente, y luego reproducirlas cuantas veces se desee.

Sin embargo, estas herramientas poseen varias desventajas: frecuentemente requieren algún tipo de conocimiento sobre desarrollo de software, usualmente incorporan actividades irrelevantes, y son muy sensibles al comportamiento, a la interfaz, a los datos y al contexto. [INTI].

2.2.1. Selenium IDE

Selenium-IDE (Integrated Development Environment) [Selenium-IDE] es una herramienta del proyecto Selenium que permite crear test-cases automáticos sobre aplicaciones Web. Es un plug-in de Firefox, lo cual constituye una desventaja en cuanto a portabilidad a otros browsers.

Dispone de un menú contextual que se abre al seleccionar un determinado elemento de la página web actual, y que permite seleccionar la acción a realizar. Esto genera scripts con una determinada sintaxis propia de la herramienta.

El pugin-in provee una Toolbar simple que se muestra a continuación.

No viene al caso la descripción detallada del funcionamiento de la herramienta, pero pueden observarse aquí los típicos botones de este tipo de aplicaciones. Uno de ellos permite la grabación de las acciones de usuario sobre la interfaz web, mientras que otros ("play") permiten la ejecución de tests individuales y de tests suites. También se puede pausar la ejecución de los tests y definir su velocidad de ejecución. Un botón de "step" permite ejecutat paso a paso un test, lo cual es muy útil para realizar debugs.

La herramienta provee un Test Case Pane en donde puede verse el script generado automáticamente para el test creado por el usuario.

Command	Target	Value
open	/	
waitForPageToLoad		
clickAndWait	xpath=id('menu_download')/a	
assertTitle	Downloads	
verifyText	xpath=id('mainContent')/h2	Downloads

Command es la acción que se desea ejecutar, Target es el objeto sobre el que se ejecutará esa acción, y Value es el valor que esa acción necesita.

Aquí puede verse que algunos commands son meramente de ejecución de una acción y otros de aserción de valores. Como ejemplo de los primeros, clickAndWait cliquea una opción y espera a la consecuente carga de la nueva página; otros comandos son de aserción de valores, por ejemplo, verifyText, y además de indicarse el Target (el elemento de la página que que contiene el texto a testear), también debe indicarse el Value, es decir el valor que se espera que tal elemento contenga.

Durante la grabación, Selenium-IDE automáticamente insertará commandos en el script del Test Case con base en las acciones del usuario. Estas acciones típicamente incluyen:

- Clickear un link – comandos *click* o *clickAndWait*
- Ingresar un valor – comando *type*.
- Selección de una opción de una lista drop-down – comando *select*
- Clickear checkboxes o radio buttons – comando *click*.

De la misma manera, también pueden insertarse al scritpt varias opciones de aserción, como por ejemplo:

- verifyElementPresent – verifica que un elemento esté presente en la página
- verifyText – verifica que el texto esperado y su correspondiente tag se encuentre en la página.
- verifyTable – verifica que una tabla tenga el contenido esperado.

Como puede observarse, esta típica herramienta "Record and Play", si bien es de gran utilidad para usuarios con conocimientos técnicos, dificulta la legibilidad de scripts a usuarios finales (no-técnicos) para entender qué es lo que hace un test. Asimismo no es portable por tratarse de un plug-in de Firefox y, como la mayoría de este tipo de herramientas, es muy dependiente de la interfaz concreta.

2.2.2. WebDriver

WebDriver [WebDriver] es una API que fue incluída en Selenium 2.0 como una de sus principales funcionalidades. WebDriver permite automatizar las acciones ejecutadas sobre un browser, brindando un mucho mejor soporte que su antecesor para páginas Web Dinámicas (en donde pueden cambiar sólo algunos de sus elementos, sin recargarse la página completa). Esta herramienta tiene como principal objetivo su uso en tests de aplicaciones web, pero éstos deben ser programados por personas con conocimiento de programación puesto que se trata de una API orientada a objetos. Independientemente de su calidad, esto último hace que no pueda ser utilizada por usuarios finales.

Selenium-WebDriver realiza llamadas directas al browser utilizando el soporte para automatización nativo de cada uno de ellos. Por lo tanto, la forma en que estas llamadas se realicen dependerá del browser que se esté utilizando. La especificación de WebDriver brinda información sobre cada uno de los "drivers de browsers".

También existe un Selenium-Server que puede utilizarse para esquemas más avanzados de testing en donde, por ejemplo, se deban distribuir tests en distintas máquinas o máquinas virtuales, o deban realizarse conexiones a máquinas remotas con distintas versiones de browsers.

La instalación y uso de este software implica la creación de un proyecto en un ambiente de desarrollo, de forma tal de poder escribir un programa utilizando Selenium. Esto dependerá del lenguaje y del ambiente de programación.

Adentrarse en los pormenores de este software (una API) carece de sentido a los efectos de este trabajo, ya que basta con saber que no puede ser utilizada por un usuario final puesto que los tests se construyen mediante un lenguaje de programción, y que deben contemplarse las diferencias de cada browser en cuanto a soporte nativo de automatización.

2.2.3. QALiber

QALiber [QALiber] permite realizar automatización de tests de una manera totalmente integrada al proceso de desarrollo de software, desde la escritura de las primeras líneas del software hasta el testeo del sistema completo. Este software es open source y free, y está compuesto por dos partes, que pueden utilizarse juntas o separadas:

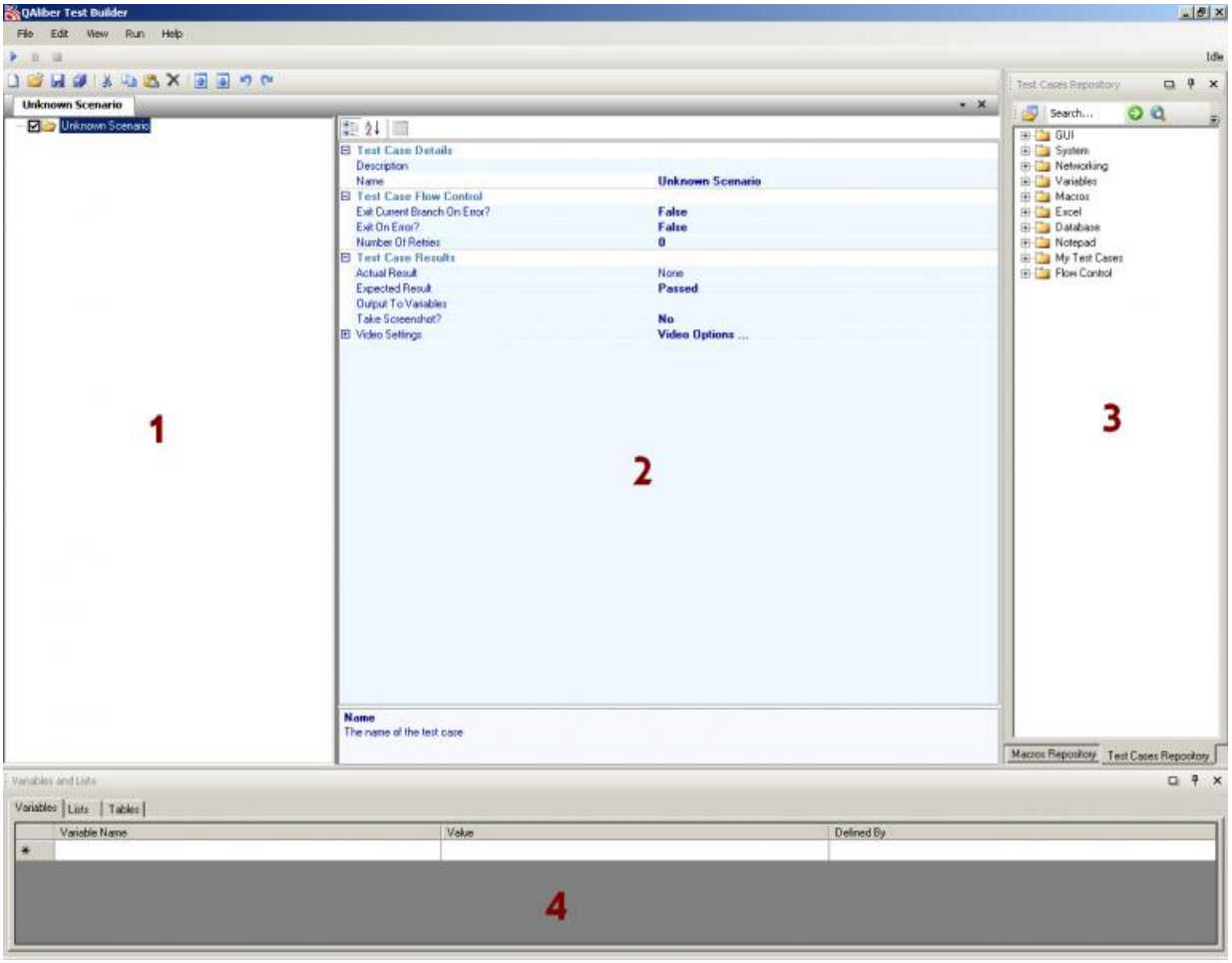
- QALiber Test Developer: es un plug-in para Visual Studio (IDE para desarrollo) y permite la creación de test como parte integral del desarrollo de software. Debido a estas características, este plug-in sólo puede ser utilizado por desarrolladores de software.

- QALiber Test Builder: es una herramienta de testing para crear y ejecutar tests sin necesidad de tener conocimientos técnicos ni de programación.

La primera opción es sólo utilizable por desarrolladores, por lo que aquí nos enfocaremos en la segunda opción ya que se trata de una herramienta amigable y que puede ser utilizada por un usuario final sin conocimientos de programación.

QALiber Test Builder es una herramienta que graba las acciones del usuario y luego las reproduce cómo si éste utilizara la aplicación. A diferencia de Selenium-IDE, no sólo puede utilizarse para aplicaciones web sino también en aplicaciones de escritorio de uso general. Las interfaces gráficas a testear no necesitan ser adaptadas con nada especial, de modo que pueden testearse aplicaciones existentes (como la calculadora de Windows).

A continuación se muestra la ventana principal de la aplicación.



Esta pantalla se compone de cuatro paneles principales:

1.-El panel de la izquierda muestra los escenarios de tests, visualizados como un árbol.
2.-El segundo panel muestra la configuración (parámetros y propiedades) del elemento seleccionado en el primer panel.
3.-El tercer panel es un repositorio de test cases y macros.
4.-El cuarto panel contiene variables, listas y tablas que permiten parametrizar los tests.

La forma de uso de esta herramienta consiste primero en grabar las acciones de operación sobre la GUI de la aplicación a testear, y luego configurar las acciones de aserción de resultados.

La primera de estas acciones, es decir, la grabación de la secuencia de acciones, no se realiza de forma contínua y fluida como en otras herramientas. Por el contrario, el usuario debe ir definiendo uno a uno los pasos individuales que conforman toda la secuencia de acción, lo cual se realiza definiendo el paso en la interfaz gráfica más arriba y, a continuación,

ejecutando el paso en la GUI de la aplicación a testear. Es decir que cada grabación coonsta de varias idas y vueltas entre la herramienta de testing y la aplicación a testear (una por cada paso simple de interacción con la GUI). En la definición de cada paso, el usuario especifica los parámetros que la GUI de la aplicación a testear necesite, como por ejemplo texto a ingresar en un campo o valor a seleccionar de una lista, entre otros. Como se mencionó anteriormente, los panes de la herramienta pueden contener juegos de datos útiles a estas necesidades.

La segunda de las acciones mencionadas, es decir, la aserción de resultados, se realiza de manera similar. Para esto el usuario selecciona algún elemento de la interfaz y define el estado en que éste deberá encontrarse, por ejemplo seleccionado, tildado o contiendo un determinado valor.

Una vez configurado todo esto, al ejecutar el test, puede verse visualmente cómo se ejecutan en secuencia la serie de pasos especificada (tal como si un usuario estuviese utilizando el Mouse y teclado), y finalmente realizando la aserción de resultados.

Con base en lo anterior puede verse que esta herramienta permite una configuración de tests sin requerir ningún tipo de conocimiento de programación. Sin embargo, es muy sensible a los cambios de interfaz y, de hecho, no puede utilizarse en momentos más tempranos de desarrollo sino que se requiere de una interfaz terminada y en funcionamiento. Por otro lado, la aplicación a testear debe tener una interfaz de tipo gráfica, ya que no puede utilizarse con otro tipo de interfaces como por ejemplo las teléfónicas.

2.2.4. Relación de estas herramientas al presente trabajo

Tal como se dijo al comienzo de esta sección, algunas de estas herramientas requieren de conocimientos técnicos para ser utilizadas, tal es el caso de WebDriver, que es una API, y QAliber Test Developer, que es un plug-in para un ambiente de desarrollo. Por otro lado, Selenium-IDE provee una forma de grabar y ejecutar acciones sobre una interfaz, pero genera un script de difícil lectura para un usuario final que necesita reconocer fácilmente qué es lo que realiza un test; además de esto, sólo puede utilizarse en interfaces Web.

QA Liber Test Liber permite la configuración de tests por parte de un usuario final de forma sencilla, pero estos tests son muy sensibles a los cambios en las interfaces gráficas. Además, sólo pueden utilizarse sobre interfaces gráficas y no de otro tipo. Finalmente, requiere que tal interfaz gráfica esté terminada, por lo que no permite automatizar tests del modelo hasta ese momento, aunque tal modelo se encuentre perfectamente finalizado.

En este trabajo se propone una solución que permitirá a los usuarios finales crear tests de manera sencilla y sin conocimientos técnicos de ningún tipo. A su vez, permitirá realizar tests sobre sistemas independientemente de su tipo de interfaz (no se centra en interfaces gráficas, web, ni desktops). Finalmente, permite testear el modelo o partes de él, aún cuando éste no tenga finalizadas sus interfaces.

2.3. Tests de Unidad

Como se ha presentado en una sección anterior, el Test de Unidad tiene como objetivo “el testeo de unidades individuales de software o hardware o grupos de unidades relacionadas” [IEEE].

La especificación de una unidad comprende un conjunto de definiciones de interfaz, y su vez, cada interfaz define un conjunto de comportamiento y de atributos de calidad que la unidad debe satisfacer. [McGregor]

En un Sistema Orientado a Objetos, estos conceptos de unidad e interfaz se perciben claramente al estar reforzados por la naturaleza propia del diseño empleado. Cada unidad puede ser una clase o un conjunto de ellas, mientras que las interfaces están constituidas por el

protocolo de mensajes que las mismas entienden.

Existen dos conceptos que son esenciales a los Tests de Unidad en contextos productivos: Automatización e Independencia de Escenarios.

2.3.1. Automatización

Los Tests de Unidad suelen estar muy ligados a los Tests de Regresión en el sentido de que los primeros no se piensan y construyen para ser ejecutados una sola vez. Por el contrario, ante cada modificación al sistema deben ejecutarse Tests de Regresión para asegurar que el release de la versión del software cumpla con las normas de calidad esperadas y con la especificación de requerimientos. En este sentido, los Tests de Regresión normalmente incluyen a los Tests de Unidad, por lo que éstos suelen ejecutarse como parte de aquellos.

Dada esta repetida ejecución de los tests, los mismos deben automatizarse lo máximo posible, tanto para minimizar el esfuerzo invertido en testing como para reducir los errores en este proceso, ejecutando los tests de forma idéntica en cada oportunidad. En la mayoría de los casos, a los Tests de Unidad se los agrupa en "Tests Suites" o "Baterías de Tests" que son ejecutados de forma automática como un conjunto.

2.3.2. Independencia de Escenarios

Es de crucial importancia que cada test de unidad se ejecute siempre de forma idéntica, dentro de exactamente el mismo contexto y bajo las mismas condiciones. De esta forma se busca asegurar que el resultado del test sea determinístico con respecto a sus datos de entrada y juegos de datos empleado, y que no depende de factores externos variables ni aleatorios.

En este sentido, para la ejecución de cada test de unidad se sigue indefectiblemente la siguiente secuencia:

1. **Set-up:** en esta etapa se prepara un contexto limpio para la ejecución del test, una de las tareas principales es la carga del juego de datos sobre el que se ejecutará el mismo.

2. **Ejecución del test:** se ejecuta el test en el escenario establecido en la etapa anterior; esta ejecución está auto-contenida en tal contexto y no se permite la comunicación con contextos externos que puedan introducir factores variables. Terminada la ejecución, se comparan los resultados obtenidos con los resultados esperados, de forma tal de determinar si la unidad testeada aprueba o no aprueba el test.

3. **Tear-down:** esta etapa es la contraparte de la primera, y su objetivo es limpiar el contexto en cuanto a los recursos utilizados durante el test (liberar conexiones, cerrar sesiones, entre otros).

Cabe repetir que la ejecución de cada test de unidad, compuesta por esta secuencia, se realiza de forma automática: al ejecutar un Test-Suite todos sus Tests de Unidad se ejecutan uno a continuación de otro. Como resultado, se genera un informe con aquellos tests que han fallado y por qué.

2.3.3. Tests de Unidad en el Modelo por Capas

En la siguiente figura se muestra la ubicación de los tests de unidad en el contexto de un sistema con dos capas: Capa de Presentación y Capa del Modelo. La primera se encarga de la

gestión de las interfaces de usuario (gráficas, móviles, telefónicas, entre otras), mientras que la segunda es la que incluye toda la lógica de negocios (entidades, reglas, procesos, restricciones). Aquella delega en ésta, quien a su vez desconoce totalmente la existencia de interfaces. Esta independencia o aislamiento es crucial para lograr altos niveles de flexibilidad.

En este contexto, los tests de unidad se utilizan para el testeo de la Capa del Modelo. Es de notar que no son parte del modelo por no ser parte del dominio del problema; por esto mismo, si bien los tests de unidad conocen al modelo, el modelo desconoce totalmente la existencia de los tests de unidad. Esto último permite "conectar" y "desconectar" de forma absolutamente directa a los tests de unidad del modelo de negocios.

En la figura se muestra a los tests de unidad "al costado" de la Capa del Modelo de negocios, que en un sistema Orientado Objetos consta principalmente de todas las clases e instancias que modelan el dominio de aplicación.

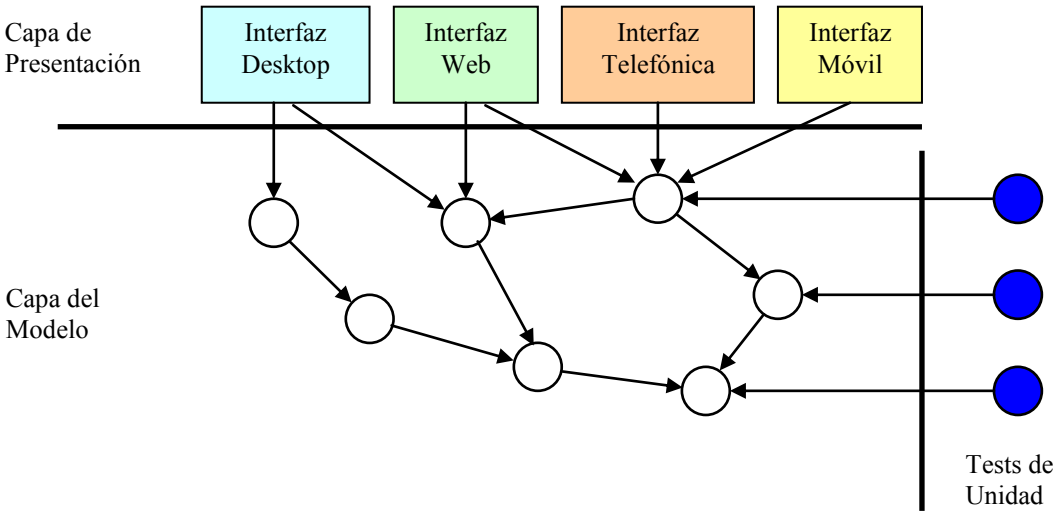


Figura 2.2 – Tests de Unidad en el Modelo por Capas

2.4. Framework para Tests de Unidad: JUnit

Existen varios frameworks para tests de unidad, por ejemplo JUnit, para Java, y SUnit, para Smalltalk. Estos frameworks, a su vez, pueden formar parte de otros más abarcativos desde el punto de vista de la Ingeniería de Software, de forma que incluyen a los Tests de Unidad como una de sus fases (por ejemplo, frameworks de Integración Contínua).

En esta sección se describirán brevemente los lineamientos generales de JUnit [JUnit]. En concreto, se explicarán brevemente las clases y métodos principales de este framework, y cómo el mismo se extiende o configura para adaptarse a un modelo de negocios en particular.

Queda totalmente fuera del alcance y del objetivo de esta sección el tratamiento pormenorizado de aspectos técnicos detallados que no hagan al entendimiento del funcionamiento en general del framework.

2.4.1. Instanciación mediante Caja Blanca

Si bien existen otras maneras, la principal forma de instanciación del Framework es mediante herencia (lo cual se conoce como instanciación White-Box, a diferencia de las Black-Box, que se realizan mediante composición). [RalphJ]

Existe una clase que es parte del framework llamada TestCase. Ésta es una clase hot-spot: se denomina así a los puntos de un framework que el usuario del mismo puede extender en relación a las particularidades de su contexto. Las instancias de esta clase entienden el mensaje run() cuyo método abarca las generalidades de ejecución de los tests de unidad en general, como por ejemplo la secuencia genérica de tres etapas descripta en una sección anterior.

La configuración del framework se realiza subclasificando la clase TestCase con la clase de test propia del usuario y, a su vez, es esta última la que conoce a la clase del modelo que se desea testear. Es decir que en este último sentido, la conexión entre las clases de tests de unidad y las del modelo de negocio se realiza mediante composición.

La subclase del usuario debe implementar uno o varios métodos de test, en cada uno de los cuales se testea una funcionalidad diferente de la unidad del modelo de negocios sujeta a prueba.

Mediante el mecanismo de Inversión de Control típico de los frameworks, JUnit, desde el "template method" [Gamma] run() de la clase TestCase enviará tres mensajes básicos de forma consecutiva: setUp(), test() y tearDown(). La subclase creada por el usuario del framework será la encargada de implementar estos tres "hook methods" de forma que se ajusten a las particularidades del modelo de negocios en cuestión.

En el método setUp(), se debe establecer el escenario de datos para la ejecución del test, entre otras cosas deben realizarse las instanciaciones de las clases del modelo de negocios que resulten de pertinencia para el test.

En el método test(), se envian mensajes a las instancias del modelo de negocios de acuerdo a la funcionalidad que se desea testear. Al final de este método deben realizarse las "aserciones", lo cual consiste en chequear que los resultados obtenidos coincidan con los esperados por el test.

Finalmente, el método tearDown() libera la utilización de recursos apropiados durante la ejecución de los dos métodos anteriores.

2.4.2. Aserciones

Tal como se mencionó en la sección anterior, en la parte final del método run() se definen las aserciones que deben chequearse para comparar los resultados obtenidos con los resultados esperados.

Estas aserciones permiten diversos tipos de comparaciones, incluso varias en un mismo método, que pueden involucrar tanto tipos primitivos como objetos.

2.4.3. Test-Suites y Dinámica del Framework

El usuario del framework podrá agrupar varios Test Cases en un Test-Suite. Al ejecutar el framework, se ejecutarán todos los tests de todos los Test Cases contenidos en el test-suite. Esto significa que antes de cada método de test de un TestCase, se ejecuta el método setUp(), y de la misma manera, posteriormente se ejecuta el tearDown().

Luego de la ejecución de todos los tests, el framework genera un informe al usuario sobre el resultado de ejecución de los tests, avisando básicamente cuáles pasaron y cuáles no.

2.5. Tests de Unidad y Capa de Persistencia

En las secciones anteriores se introdujo brevemente el Modelo por Capas enfocándonos básicamente en dos de ellas: Capa de Presentación y Capa del Modelo; sin embargo, el modelo más utilizado en ambientes productivos consta de al menos tres capas, por lo que a las dos anteriores se le suma una inferior: la Capa de Persistencia [PersOO].

Esta última capa, en un ambiente Orientado a Objetos, se encargará básicamente de gestionar el almacenamiento de los objetos de la Capa del Modelo.

Siguiendo la misma filosofía ya explicada, cada capa conoce a la inferior pero no debe ocurrir lo contrario, de forma tal de maximizar la flexibilidad y el desacoplamiento. En este sentido, la Capa del Modelo conocerá a la Capa de Persistencia pero no ocurrirá la inversa; por otro lado, la Capa de Presentación es totalmente ajena a la de Persistencia.

En la siguiente figura se muestra la extensión del modelo por capas ya propuesto, ahora extendido con esta nueva capa. Como puede observarse, los tests de unidad siguen enfocándose en la Capa del Modelo, ya que es la que implementa el dominio del negocio y sus reglas.

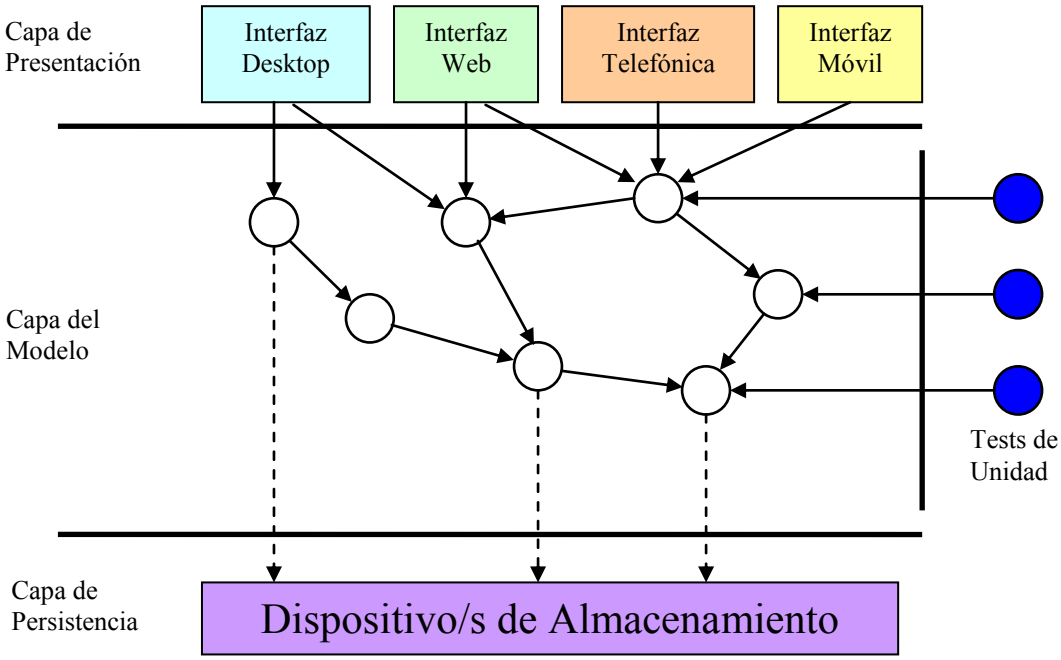


Figura 2.3 – Tests de Unidad y Capa de Persistencia

2.5.1. Implementación de la Capa de Persistencia

La implementación de la capa de persistencia en ambientes orientados a objetos puede realizarse de distintas maneras. Las más utilizadas son aquellas que utilizan Bases de Datos, y a su vez, esto puede realizarse de varias maneras, entre las que podemos destacar dos: bases de datos orientadas a objetos y bases de datos relacionales.

Las bases de datos orientadas a objetos permiten mantenernos dentro del mismo paradigma de la Capa del Modelo, lo cual evita el llamado "mismatch de impedancia". En este sentido, son muy transparentes al desarrollador ya que, en general, permiten al mismo pensar sólo en términos de los objetos de la capa del modelo.

No obstante lo anterior, en la industria aún hoy tienen mucho arraigo las bases de datos relacionales, y son éstas las que finalmente suelen utilizarse para implementar la capa de persistencia. Sin embargo, en este caso hay que solucionar el salto de paradigmas que hay entre la Capa del Modelo (orientada a objetos) y la de Capa de Persistencia (relacional). Mientras el desarrollador diseña e implementa la Capa del Modelo debería pensar sólo en términos de objetos, de ninguna manera en cómo estos se almacenan y mucho menos en cómo estos pueden descomponerse en partes y traducirse para ser almacenados en una base de datos

relacional. Para solucionar esto último, en la industria se suele incluir una nueva capa, a la que se considera "secundaria" o "traductora" entre las dos capas mencionadas, consideradas de mayor jerarquía. Éstos son los Mapeadores Objeto-Relacional y serán explicados brevemente en la próxima sección.

Finalmente, cabe mencionar que existen otros mecanismos de persistencia diferentes a las bases de datos, pero no se los suele utilizar como almacenamiento principal de sistemas, como por ejemplo las Bases de Datos XML.

2.5.2. Mapeadores Objeto-Relacional - Hibernate

Tal como se introdujo en la sección anterior, los Mapeadores O/R se utilizan cuando la Capa del Modelo es Orientada a Objetos y la Capa de Persistencia es Relacional, típicamente una Base de Datos. El Mapeador O/R se ubica entre estas dos capas y realiza las "traducciones" necesarias entre ambos paradigmas; de esta manera, el desarrollador de la Capa del Modelo se abstrae de forma completa (idealmente) de los aspectos de persistencia.

Luego de algunas configuraciones iniciales, el mapeador se encargará de descomponer los objetos del modelo para almacenarlos en las tablas propias del paradigma estructurado y relacional. Asimismo, también se encargará de la recuperación de la información residente en la capa de persistencia para recomponer los objetos en la capa del modelo, y dar la sensación al desarrollador de que los objetos "siempre están ahí".

Estos mapeadores también se encargan de muchas otras tareas importantes, como por ejemplo el manejo del buffering o la traducción de consultas en OQL (lenguajes de consulta orientados a objetos) a SQL, que es el que finamente el sistema utiliza en la capa de persistencia.

Existen varias implementaciones de mapeadores disponibles en la industria; uno de los más utilizados es Hibernate [Hibernate]. Dado que está fuera del alcance de este trabajo la descripción pormenorizada de esta tecnología, sólo mencionaremos algunos aspectos básicos de la misma.

Hibernate permite mapear un modelo OO a varias marcas de bases de datos, es decir que no se atan a empresas particulares; esto permite que pueda cambiar incluso la base de datos que se utiliza.

Permite mapear clases individuales o jerarquías completas a las tablas del modelo relacional de la capa de persistencia; para esto existen diversas técnicas que no se desarrollarán aquí. Una ventaja de Hibernate es que este mapeo inicial puede ser hecho por una persona o automáticamente por la misma tecnología, permitiendo luego cambiar aquellas partes del mapeo cuando se lo considere necesario (por razones de performance, por ejemplo). Estos mapeos de clases se realizan en archivos XML, siguiendo una estructura estricta y predefinida en un DTD. Finalmente, en cuanto a configuración, existe un archivo XML de configuración principal, que es el que establece los parámetros de conexión a la Base de Datos y también referencia a los archivos XML de mapeo previamente mencionados que deban contemplarse.

Por último, cabe mencionar el lenguaje HQL (Hibernate Query Language) que es el lenguaje de consultas orientado a objetos utilizado por el desarrollador en la Capa del Modelo. Hibernate traduce de forma automática las consultas en HQL a consultas en SQL, que son las que entiende la capa de persistencia; a la inversa, con los resultados de esta consulta, Hibernate genera las instancias con la información recuperada y presenta los objetos en la Capa del Modelo, dentro del paradigma esperado.

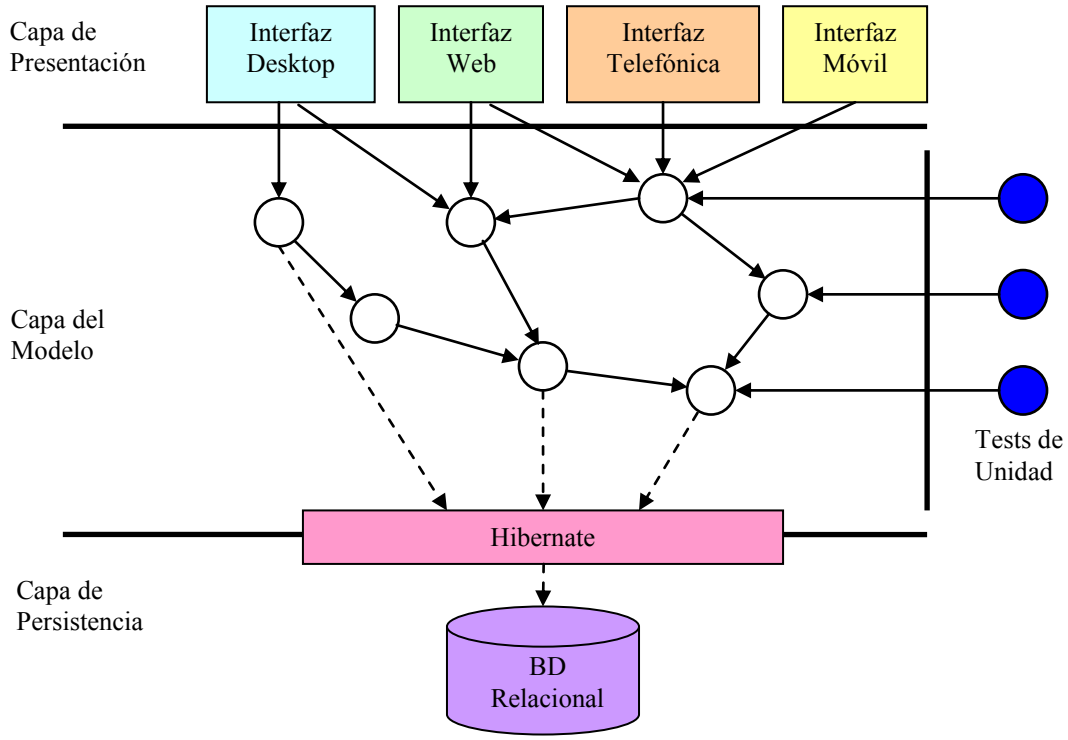


Figura 2.4 – Mapeo Objeto/Relacional mediante Hibernate

2.5.3. Problemas del Esquema Planteado

En las secciones anteriores se han descripto tanto el modelo por capas como algunas tecnologías para implementarlo, incluyendo también conceptos y tecnologías utilizadas para los tests de unidad. A continuación se describen algunos problemas o limitaciones que esta arquitectura y tecnologías tienen con respecto al diseño y ejecución de los tests de unidad.

- Juegos de Datos: Los tests de unidad deben ser programados por los desarrolladores, e incluso deben programar y preparar los juegos de datos a utilizar. Esto último les agrega un trabajo extra que, además, suele ser tedioso cuando involucra un gran volumen de datos (si bien el cliente final es quien puede proveer los datos, el programador debe incorporarlo al test).

- Ambiente: La mayoría de los frameworks actuales, como JUnit, suelen funcionar muy bien cuando se ejecutan en memoria, ya que resulta fácil crear y limpiar el contexto (escenario) de ejecución. No obstante lo anterior, prácticamente todos los sistemas comerciales almacenan su información en Bases de Datos, lo cual genera la necesidad de poder integrar estas BDs en los tests de unidad. Y es en esto último en donde aparece el mayor problema: al incorporar este tipo de persistencia, no resulta sencillo realizar y asegurar la limpieza e independencia de escenarios (ahora incluyendo bases de datos) entre una y otra ejecución de los tests.

- Test completo: JUnit permite realizar un conjunto amplio de aserciones pero siempre de baja granularidad. No provee herramientas para corroborar que un gran conjunto de datos es igual a otro gran conjunto de datos. Si bien esto no es usual

en los tests de unidad, puede resultar útil para asegurarse que la ejecución de un test no produjo efectos laterales en una región bien amplia de información del sistema.

2.6. Framework para Tests de Unidad con Persistencia: DBUnit

Existen frameworks que intentan aportar soluciones a los problemas descriptos en la sección anterior. Por ejemplo, "DBUnit" es una extensión al framework "JUnit" para incorporar a aquél la funcionalidad de Tests de Unidad con Bases de Datos.

En esta sección se describirán los aspectos básicos del diseño y uso del framework, mientras que en un anexo de este trabajo pueden encontrarse descripciones y ejemplos al máximo nivel de detalle sobre su configuración y uso.

2.6.1. Aspectos Básicos

El objetivo de DBUnit [DBUnit] es proveer las características y seguridades de los tests de unidad, tal como se ha descripto anteriormente, en el contexto de sistemas industriales con fuerte soporte de Bases de Datos relacionales en su Capa de Persistencia e implementación en Java en su Capa del Modelo. En este sentido, asegura que los tests se realizan sobre un escenario de bases de datos idéntico y limpio en cada ejecución, automatizando la generación de tal escenario a partir de un juego de datos determinado.

DBUnit puede utilizarse con una amplia variedad de bases de datos, entre ellas MySQL y Oracle. Su forma de utilización es independiente de la base de datos concreta, con lo cual ésta podría cambiarse por otra sin afectar la implementación de los tests.

El usuario del framework establecerá una única vez el juego de datos a utilizar como escenario en cada ejecución de los tests. El framework automáticamente cargará tales datos en la base de datos antes de cada test y, como contrapartida, luego del test se asegurará de limpiar los datos utilizados; con esto queda claro que cumple con la secuencia de tres pasos inherente a todo test de unidad.

2.6.2. Especificación del Escenario de Pruebas

Como se dijo en la sección anterior, el usuario del framework debe especificar el juego de datos que se utilizará como base para cada test. Esta especificación se realiza mediante un archivo XML y debe cumplir con una estructura precisa establecida por el framework en un archivo DTD.

Si bien este archivo XML puede ser construído manualmente o mediante herramientas tradicionales de gestión de formato XML, el framework provee la ventaja de poder generarlo como una extracción de la base de datos. En este sentido, pueden definirse los datos de la base de la forma que se considere adecuada (por ejemplo mediante las interfaces gráficas de la misma aplicación) y luego generar el xml a partir de allí.

Este archivo deberá ser colocado en el file system para ser automáticamente levantado y cargado a la base de datos antes de cada test.

2.6.3. Instanciación mediante Caja Blanca

La forma de instanciación del framework es muy similar a la de JUnit, de hecho, de alguna manera tal instanciación puede verse como una continuación de la de JUnit. En concreto, DBUnit provee una clase llamada DBTestCase, que es subclase de TestCase, provista por JUnit. Para instanciar el framework el usuario debe crear su propia clase de test ubicándola

como subclase de DBTestCase; de esta manera, la clase de test creada heredará todo el comportamiento necesario tanto de JUnit como de DBUnit.

La clase creada deberá implementar un "método hook" llamado getDataSet(), que se encargará de tomar desde el file system el juego de datos creado en el punto anterior (dataset.xml). La carga inmediata posterior del mismo en la base de datos es exclusiva responsabilidad del framework, por lo que es totalmente transparente al usuario. De la misma manera que en JUnit, una clase de test puede tener varios métodos de test, por lo que esta operación de carga del juego de datos se llevará a cabo antes de la ejecución de cada uno de ellos.

2.6.4. Ejecución de Tests y Aserciones

Los métodos de test serán ejecutados uno tras otro, en secuencia y siempre cargando el juego de datos limpio contenido en el dataset.xml. En el método de testing pueden realizarse todas las acciones que pueden realizarse en JUnit (por herencia) y además otras referentes a bases de datos, en particular en relación a las aserciones.

Pueden crearse o levantarse instancias de la base de datos de forma transparente (por ejemplo mediante Hibernate), y ejercitarlas enviándoles mensajes, lo cual provocará que se modifiquen también otras instancias del juego original de la base de datos.

Al final de la ejecución, como en todo test, deben realizarse las aserciones que comparen los resultados obtenidos con los esperados. Aquí hay dos posibilidades. En primer lugar pueden realizarse todas las aserciones al igual que en JUnit; en este caso, de DBUnit estaríamos utilizando sólo la funcionalidad de establecimiento automático de grandes escenarios, lo cual no es poco y puede resultar suficiente para muchos casos.

Otra posibilidad es realizar aserciones más generales, que comparen todo el estado de la base de datos o al menos una parte de la misma. Esto último puede ser útil cuando los resultados a comparar son muchos o bien cuando se quiere asegurar que, como efecto lateral, no se ha modificado ningún dato de una zona de la base de datos. Cabe destacar que en esta segunda alternativa, se pierde la división de paradigmas entre las Capas del Modelo y de Persistencia, ya que desde las clases de test se estaría conociendo la forma en que los objetos son descompuestos en tablas relacionales.

2.6.5. Problemas del Esquema Planteado

Si bien la metodología planteada en esta sección provee grandes ventajas, como la incorporación automática de la capa de persistencia en los tests y la verificación de áreas completas de datos, aún subsiste el problema de la creación de los tests y juegos de datos.

Esto último refiere a que los tests de unidad aún deben ser programados por los desarrolladores, incluso la preparación de los juegos de datos a utilizar (aunque ahora ya no programados, sino en archivos .xml). Si bien DBUnit reduce este esfuerzo, aún se necesita de un programador que realice estas tareas.

El modelo que se presenta en los próximos capítulos intenta resolver este problema, delegando completamente al usuario final tanto la creación de tests como de juegos de datos.

3. SISTEMA FINANCIERO CUENTA ÚNICA DEL TESORO

En este capítulo se describen, de forma general, algunos aspectos actuales de la situación en la Tesorería General de la Provincia que se toman como base para el desarrollo de este trabajo. Esto incluye el modelo financiero utilizado en provincia (CUT), el sistema informático con el cual se la da soporte y algunas prácticas actuales para el testing de tal sistema.

En capítulos posteriores se describirán en detalle los desarrollos que este trabajo propone sobre la base de la situación actual descripta en este capítulo.

3.1. Modelo Financiero Cuenta Única del Tesoro

El proyecto Cuenta Única del Tesoro se encuadra dentro de la reciente Ley de Administración Financiera Nro. 13767 [Ley13767]. Si bien a los efectos de este trabajo no es necesario conocer los detalles del dominio financiero, en esta sección se brinda una muy breve descripción del mismo.

3.1.1. Introducción al Modelo Financiero CUT

El concepto financiero del Sistema de Cuenta Única de Tesoro (SCUT) está basado en el mantenimiento de una única Cuenta Corriente bancaria (CUT) en el Banco de la Provincia, operada exclusivamente por la Tesorería General de la Provincia (TGP). Hacia esa cuenta deben dirigirse todos los recursos recaudados por los organismos provinciales.

Esta es la diferencia esencial con el modelo financiero tradicional, en donde cada organismo provincial posee sus propias cuentas con sus propios fondos, éstos quizás en desuso temporal, lo que impide o dificulta a la Tesorería General de la Provincia hacer uso de los mismos para afrontar otros gastos de urgencia, con el compromiso de reponerlos luego, por supuesto.

Dado que todos los recursos recaudados son administrados en la CUT por la TGP, ésta debe funcionar como una administración centralizada de fondos, concentrando de esta manera los recursos disponibles y conociendo en detalle a quién pertenece cada recurso dirigido a la CUT.

Para esto, los Organismos deben mantener en el Banco Provincia un conjunto de Cuentas Bancarias exclusivamente a los efectos de recibir fondos en concepto de recaudación de recursos. Estas son las llamadas Cuentas Recaudadoras Bancarias.

Para el funcionamiento de la Tesorería bajo el modelo de gestión CUT se desarrolla un sistema de Cuentas Escriturales contables que identifican la porción de fondos que le corresponden a cada organismo del total residente en la CUT. Es por esto que estas Cuentas Escriturales son subcuentas contables de la CUT y se actualizan al registrarse transacciones de los Organismos y de la TGP en el sistema, con lo cual la sumatoria de los saldos de dichas Cuentas Escriturales conciliará con el saldo de la Cuenta Única del Tesoro.

La TGP suministra a los Organismos extractos de sus Cuentas Escriturales para que éstos puedan realizar sus procesos de Conciliaciones Jurisdiccionales y observar los ingresos y egresos relacionados a las mismas.

Además de la Gestión de Ingresos, la TGP es responsable de llevar a cabo la Programación de Financiera (de caja) de la CUT, lo cual dará lugar a la posterior Gestión de Pagos.

Estos pagos serán efectivizados por la TGP, a pedido de los Organismos, y serán realizados con fondos de la CUT lo cual, además, generará un débito en la Cuenta Escritural contable de origen del Organismo pagador.

Corresponde también a la TGP gestionar el proceso Registro de Ingresos de Recursos a la CUT como así también el de Conciliación de Pagos.

Con base en lo descripto anteriormente, la gestión general del modelo financiero CUT consta de cuatro sub-gestiones principales:

- Gestión de Ingresos
- Gestión de Pagos
- Gestión de la Conciliación
- Gestión de la Planificación Financiera

Queda fuera del alcance de este trabajo la descripción detallada de cada una de estas gestiones, puesto que ahondaría en temas económico-financieros que no hacen a la esencia del problema que se pretende abordar aquí.

3.1.2. Diagrama Básico

El diagrama del modelo de gestión, en su visión más global, se muestra a continuación.

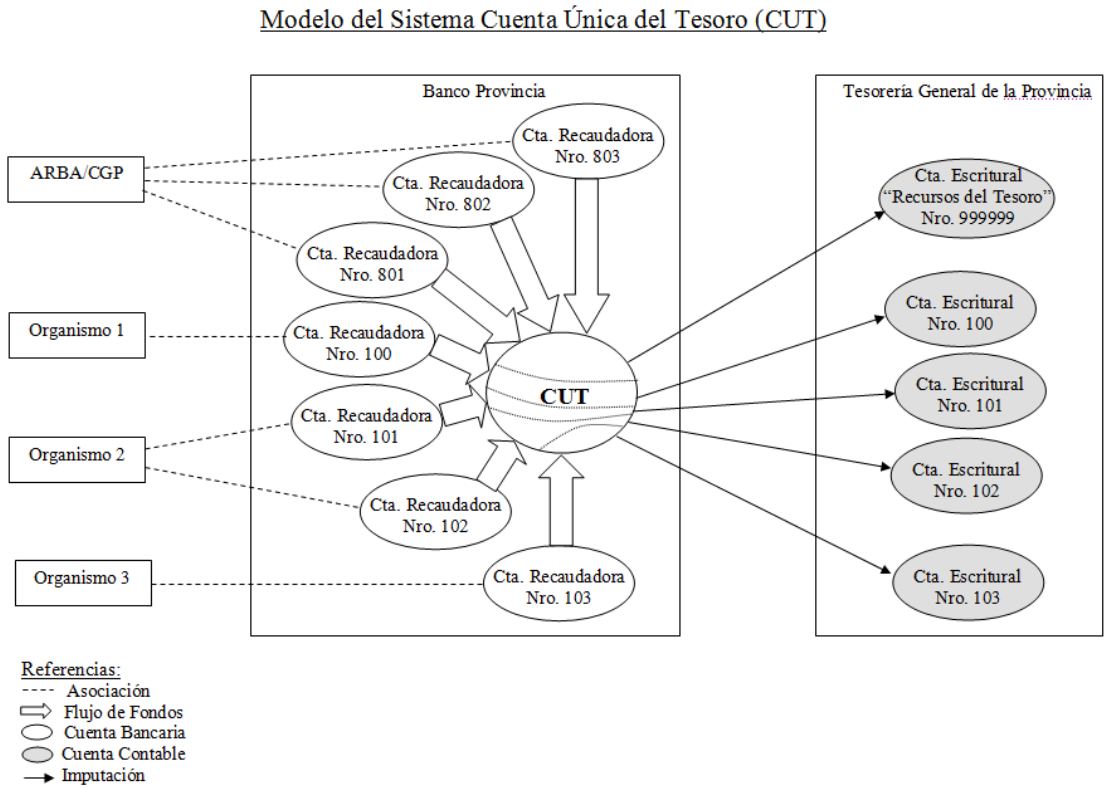


Figura 3.1 – Modelo Financiero Cuenta Única del Tesoro (CUT)

3.1.3. Descripción de las Entidades del Modelo Financiero

En esta sección se describen las entidades y relaciones del diagrama anterior. En la

primera parte se describe el concepto y naturaleza de las entidades mientras que en la segunda se describen los aspectos dinámicos de interrelación entre las mismas.

3.1.3.1. Entidades y Relaciones

- Organismos: son tanto organismos provinciales de la Administración Central como descentralizados.

- Cuentas Recaudadoras: son cuentas recaudadoras bancarias que poseen los organismos en el Banco Provincia. En estas cuentas los organismos perciben sus ingresos por recaudación y no pueden ser utilizadas para efectuar pagos.

- CUT: Es la Cuenta Única del Tesoro, una cuenta bancaria en pesos en el Banco Provincia en donde residen los fondos del Tesoro y de los Organismos Provinciales.

- Cuentas Escriturales: son cuentas contables administradas centralizadamente por la TGP. Cada organismo tendrá asociadas tantas Cuentas Escriturales como Cuentas Recaudadoras posea, y en ellas la TGP imputará contablemente créditos y débitos según cada uno registre ingresos o egresos de fondos de la CUT.
  Cada Cuenta Recaudadora Bancaria de un organismo tendrá como contraparte contable a una Cuenta Escritural en la TGP. En la sub-sección siguiente se detalla el funcionamiento conjunto de ambos tipos de cuentas.

- Extractos de Cuentas: tanto para las Cuentas Recaudadoras bancarias como para las Cuentas Escriturales se generarán extractos que detallarán cada movimiento ocurrido en ellas. A los extractos de las primeras los generará el Banco Provincia, mientras que a los extractos de las últimas los generará la TGP.

3.1.3.2. Flujos de Fondos e Imputaciones

- Un organismo puede tener una o varias Cuentas Recaudadoras Bancarias en el Banco Provincia para recaudar por distintos conceptos.

- Cada Cuenta Recaudadora Bancaria tiene asociada una Cuenta Escritural (contable) en la TGP. Las Cuentas Escriturales permiten su identidad y relación con la Cuenta Recaudadora del organismo. Algunas cuentas recaudadoras tienen como contraparte una única Cuenta Escritural, la cual representa los "Recursos del Tesoro".

- Diariamente los organismos perciben sus ingresos por recaudación en las Cuentas Recaudadoras Bancarias. Al final del día, el Banco Provincia vacía todas estas cuentas transfiriendo la totalidad de los montos a la CUT (el saldo de todas las Cuentas Recaudadoras quedará en cero).
  Luego de realizado este procedimiento, el banco envía a la TGP el extracto bancario electrónico de la CUT (en el cual se identifica la procedencia de cada ingreso y el monto, entre otros).

- Como consecuencia del procedimiento anterior, al día siguiente la TGP encuentra fondos en la CUT aún no registrados contablemente en las Cuentas Escriturales, por lo que procede a realizar de forma automática tales registros.
  Para esto la TGP toma el extracto bancario electrónico de la CUT remitido por el banco y, mediante un proceso informático automatizado, procesa el mismo identificando los ingresos a la CUT (monto y Cuenta Recaudadora de origen) e imputando contablemente en

cada Cuenta Escritural correspondiente.
Como resultado de este proceso, se tiene registrada contablemente la subdivisión del saldo total de la CUT entre los distintos organismos y los Recursos del Tesoro.

- Diariamente cada organismo recibe, por parte de la TGP, el extracto electrónico de su Cuenta Escritural, que podrá utilizar para su control. De la misma forma también recibe, por parte del Banco, los extractos de sus Cuentas Recaudadoras Bancarias para conocer el detalle de los ingresos.

- Los pagos que deben afrontar los organismos se realizan con los fondos que se disponen en la CUT. Tales pagos debitan fondos de la CUT e imputan contablemente el débito en la Cuenta Escritural que se haya utilizado para imputar el pago. Los organismos puedenn corroborar tales pagos en el extracto de su Cuenta Escritural, remitido por la TGP, cuando así corresponda.

3.2. Sistema Informático Actual (SCUT)

Para dar soporte a la gestión del modelo financiero descripto en las secciones anteriores, la TGP utiliza actualmente el sistema informático SCUT. Si bien gran parte de este sistema ya se encuentra en funcionamiento, existe funcionalidad que continúa en desarrollo, por lo que aún se necesita un gran esfuerzo de testing.

Este sistema informático contempla cuatro módulos que se enfocan en las cuatro gestiones financieras que se han descripto más arriba, más un módulo administrativo que permite gestionar las entidades, usuarios, permisos y toda la información administrativa y de base que los cuatro módulos financieros necesitan para funcionar.

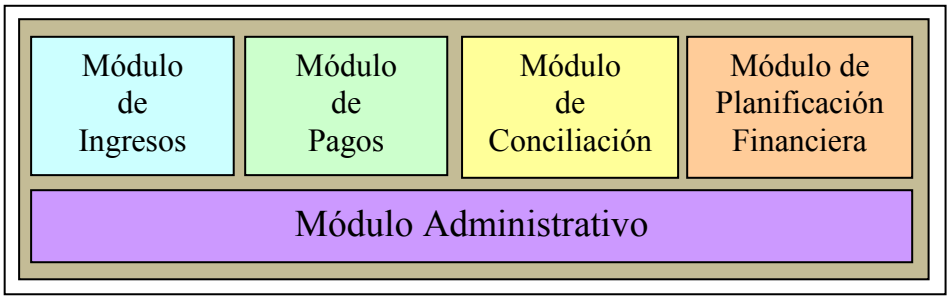


Figura 3.2 – Sistema Informático SCUT

En las secciones siguientes se describen brevemente cada uno de estos módulos informáticos desde el punto de vista funcional, mientras que en la última de ellas se describen los aspectos técnicos y tecnologías principales de su construcción.

3.2.1. Módulo de Ingresos

Este módulo posibilita el registro de todos los recursos que ingresan a la CUT que se originan por recaudación, depósitos, ajustes u otros tipos de ingresos enmarcados en la Ley Financiera Provincial.
Tal como se explicó anteriormente, todo ingreso es registrado en el Sistema CUT a partir de una Cuenta Recaudadora Bancaria, cualquiera sea su origen. En este sentido el sistema SCUT provee entidades que representan a estas cuentas, tanto bancarias como escriturales, y que registran los movimientos contables antes descriptos, entre otros.

El sistema informático permite realizar tales registros de dos posibles maneras: automática o manual. La forma automática se basa en el parseo del extracto electrónico de la CUT que, como se describió más arriba, el banco envía a la TGP. De esta manera, el sistema consulta las asociaciones que tiene pre-cargadas entre cuentas bancarias recaudadoras y cuentas escriturales y, en base a esto, imputa automáticamente el ingreso la cuenta escritural correspondiente. De no poder establecerse esta asociación automáticamente (por la razón que fuere) durante el procesamiento del extracto elecetrónico, el sistema registra al movimiento como "Recurso No Identificado", y lo deja pendiente para que el usuario, posteriormente, detecte el destino del mismo y lo impute manualmente en la cuenta escritural que corresponda.

3.2.2. Módulo de Pagos

La Gestión de un Pago no sólo consta de la efectivización del mismo sino de un proceso más complejo y largo del cual tal efectivización constituye sólo su eslabón final. El sistema informático SCUT contempla tal proceso completo, que involucra las siguientes gestiones:

- Recepción de Expedientes de Pago
- Gestión de Retenciones Impositivas
- Gestión de Medidas de Afectación Patrimonial
- Generación y Gestión de diversos Medios de Pago.
- Efectivización de Pagos Parciales y Totales
- Gestión de Anulaciones y Devoluciones

Además de los Medios de Pago tradicionalmente empleados, en el Modelo Financiero CUT se cuenta con un nuevo tipo de pago que puede ejecutarse entre aquellos organismos que estén incorporados y registrados en el sistema.

Este nuevo tipo de pago, denominado "Pago por Transferencia", se encuentra implementado en el sistema informático SCUT, y consiste en un traspaso entre Cuentas Escriturales registrales de organismos que operan en el Sistema de Cuenta Única. Es por esto que se destacan por ser un tipo de operación que no genera transacción bancaria propiamente dicha (por lo cual no afecta el saldo de la CUT) sino que sólo genera movimientos compensatorios entre las Cuentas Escriturales de los Organismos involucrados en la operación, a través de un asiento de crédito en el que recibe y de débito en el que paga.

3.2.3. Módulo de Conciliación

Mediante este módulo la TGP realiza un control sistemático de los movimientos de ingresos y egresos en las cuentas del Sistema CUT, tanto en las Cuentas Recaudadoras Bancarias y en las Cuentas Escriturales registrales como en la misma CUT.

Una de las formas fundamentales mediante las que se realiza este control es la Conciliación de Pagos, en la cual la TGP verifica diariamente los pagos realizados contra el correcto registro en las Cuentas Escriturales correspondientes.

Este proceso de Conciliación de Pagos se complementa con un proceso de Registro de Ingresos, mediante el cual se realizan verificaciones e imputaciones contables en las Cuentas Escriturales, tal como se describe en secciones anteriores.

Tanto el procesos de Conciliación de Pagos como el de Registro de Ingresos se realizan de forma automática, para lo cual se considera a los Extractos Bancarios Electrónicos (remitidos diariamente por el Banco Provincia) como una de las entradas esenciales al sistema informático. Asimismo, ambos procesos cuentan también con un módulo que permite realizar una gestión manual, y que se utilizan sólo para los ítems puntuales en los que la gestión automática no haya podido ser llevada a cabo por alguna razón.

3.2.4. Módulo de Planificación Financiera

La Planificación Financiera trata de las previsiones a corto y largo plazo de los ingresos y egresos que deberá enfrentar la TGP, de forma de planificar estrategias para gestionarlos. A la planificación a corto plazo de esta gestión se la denomina Programación de Caja, y determina un plan ejecutivo para la gestión de los movimientos de entrada y salida de la CUT que se darán a la brevedad.

Este módulo contempla la Programación de Caja, es decir la programación de la Cuenta Única del Tesoro (CUT). Para lo cual se contemplan diversos aspectos de la gestión, por ejemplo, para determinar los pagos a efectuar, se deben tener en cuenta la disponibilidad en las Cuentas Escriturales involucradas, el Nivel de Exigibilidades, las Autorizaciones Efectuadas, diversa Información sobre Pagos y datos provenientes de la Planificación Financiera global.

3.2.5. Módulo Administrativo

Este módulo permite gestionar todas las entidades e información de base que los módulos descriptos anteriormente necesitan para funcionar. Esto incluye tanto información de gestión de sistema como información financiera. Como ejemplo del primer tipo se pueden citar la gestión de usuarios, permisos, perfiles y configuraciones personales, mientras que como ejemplos del segundo tipo de información se pueden citar el ABM de entidades de dominio básicas, como cuentas escriturales, organismos y cuentas bancarias recaudadoras.

3.2.6. Aspectos técnicos y Tecnologías utilizadas

Al nivel técnico el sistema SCUT se compone de diversas capas, tres de las cuales son las mencionadas previamente en este documento: Presentación, Modelo y Persistencia.

En la Capa de Presentación, las interfaces son clientes ricos realizados en Eclipse RCP (Rich Client Protocol), que es un framework para la creación de GUIs en Java. La Capa del Modelo es Orientada a Objetos, contiene la representación de las entidades financieras presentadas en la sección anterior (entre muchas otras) y está realizada también en Java, mientras que la capa de Persistencia está implementada mediante una base de datos Oracle. Tal como se describió anteriormente, el salto de paradigmas entre las capas de modelo y persistencia se salva mediante un Mapeador O/R, que en particular es Hibernate.

3.3. Tests de Unidad en el Contexto Actual del Sistema CUT

En esta sección se describe la forma en que actualmente se confeccionan los tests de unidad para testear el sistema SCUT, como así también algunas mejoras que podrían realizarse en esta práctica, las cuales son las que constituyen el objetivo de este trabajo.

Para el testeo del sistema, entre otros, se utilizan Tests de Regresión y de Unidad. Los primeros se realizan de forma manual, mientras que los tests de unidad son confeccionados mediante JUnit por los desarrolladores en base a juegos de datos pequeños, y sugeridos por el equipo de analistas funcionales. En consecuencia, los programadores deben invertir una carga de tiempo y esfuerzo en la generación, procesamiento y carga de estos juegos de datos.

Estos tests de unidad se ejecutan básicamente en memoria y no hacen uso de la capa de persistencia. Asimismo, las aserciones se realizan sobre unos pocos objetos y nunca sobre un área más amplia del dominio de aplicación para detectar efectos laterales más lejanos, aunque éstos fueran críticos.

DBUnit daría una solución parcial a estos problemas. En primer lugar, se podría pasar

mucha carga desde los desarrolladores hacia el usuario final en cuanto a la elaboración de los juegos de datos a utilizar en cada test; esto es aplicable en estados del proyecto avanzados, como el actual, en donde el usuario final ya dispone de una GUI madura mediante la cual puede generar los datos necesarios. Una vez hecho esto, sólo debe dar aviso a los desarrolladores para que éstos disparen el exportación automática de la base de datos hacia un xml que contendrá el juego de datos base: el dataset. De esta manera, los desarrolladores no emplean tiempo y esfuerzo en esta generación, sino que sólo se preocupan en la parte de ejecución del test y las aserciones correspondientes, dos tareas mucho menores en tamaño.

Esto no sólo permite una distribución de tareas que alivia al equipo de desarrollo, sino que provee un ambiente "políticamente propicio" a la aceptación del producto final, ya que los tests se ejecutan sobre el juego de datos que el usuario que debe aceptar el producto ha generado por sí mismo.

Por otro lado, además de las aserciones típicas de JUnit sobre objetos puntuales del modelo, DBUnit permitirá realizar aserciones más amplias, comparando volúmenes de datos más grandes de forma de asegurar que zonas completas del modelo no sufren ningún tipo de efecto lateral y quedan tal cual se espera luego de la ejecución del test.

Si bien DBUnit provee las mejoras descriptas, existen otros problemas que aún quedan por resolver. El más importante es que aún se necesita la inversión de esfuerzo por parte de los desarrolladores en el procesamiento y carga de los juegos de datos definidos por el usuario, como así también la programación misma de los tests.

El presente trabajo propone una solución integral para estos problemas.

4. ESQUEMA GENERAL DE TESTING

En este capítulo se describe el Esquema General de Testing y la dinámica básica de su uso por parte de un usuario final. Se presentan las dos componentes principales de tal esquema, la Aplicación de Testing y la Aplicación Objetivo. Esta última no constituye una aplicación fija y determinada, sino que su lugar lo ocupa cualquier aplicación que se desee testear mediante este esquema. Este dinamismo de configuración de la Aplicación Objetivo (aplicación a testear) determina ciertas particularidades técnicas que ésta debe cumplir y que se presentan en la primera sección de este capítulo.

4.1. Arquitectura de la Aplicación Objetivo

El modelo de testing propuesto en este trabajo, cuyo objetivo es dar solución al último problema planteado, consiste en una continuación de la evolución de la línea de soluciones descriptas anteriormente. En este sentido, para poder comprender el próximo paso de la evolución que constituye la solución propuesta, es necesario comprender antes la arquitectura general de las aplicaciones a testear (aplicaciones objetivo). En esta sección se describirán las capas básicas de esta arquitectura y sus componentes elementales.

4.1.1. Arquitectura básica por capas

La aplicación a testear debe cumplir con una determinada arquitectura, que sin embargo es una muy utilizada en la industria actualmente. En sus términos más abstractos, es muy similar a la ya expuesta en secciones anteriores. A continuación se la muestra nuevamente, ahora sin incluir los tests de unidad.

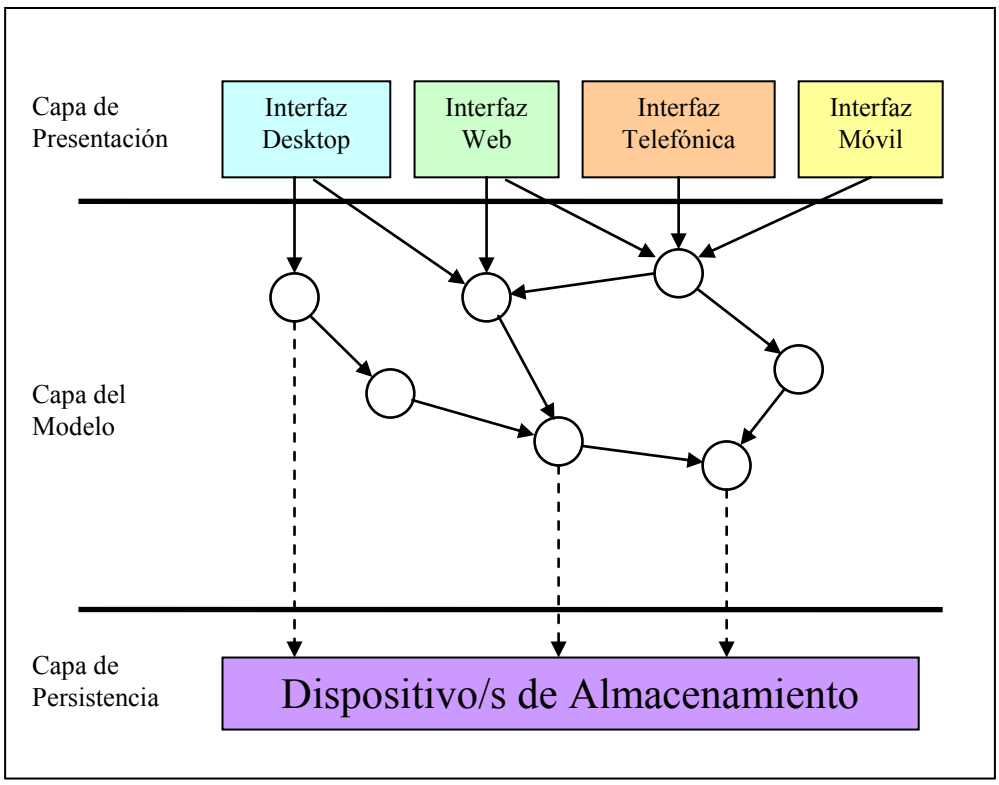


Figura 4.1 – Arquitectura por Capas Básica

El objetivo de la división por capas y la responsabilidad de cada una ya han sido descriptos previamente.

En un paso más hacia una arquitectura más concreta, ya deben definirse tecnologías para estas capas. Como se ha explicado anteriormente, las tecnologías de la aplicación objetivo para la que se diseña el modelo de testing son Java en la capa del modelo, una Base de Datos relacional para la capa de persistencia y Hibernate como mapeador entre estas capas. Con base en esto, la arquitectura anterior aparece de forma más concreta como se muestra a continuación.

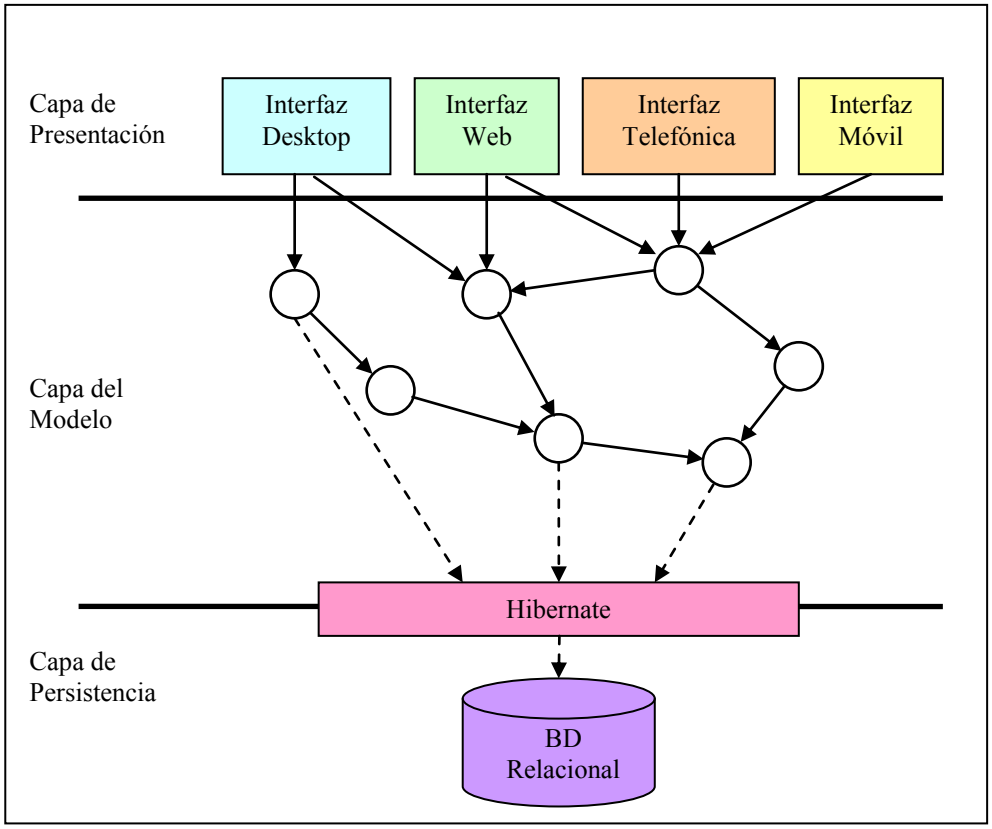


Figura 4.2 – Capa de Persistencia mediante BD Relacional y Hibernate

Esta arquitectura también ha sido explicada anteriormente, por lo que el lector puede remitirse a tal explicación. Aquí sólo se han eliminado los tests de unidad, ya que no corresponden al modelo y se resolverán de otra manera.

4.1.2. Arquitectura SOA

En la figura anterior puede observarse que las distintas interfaces de usuario de la capa de presentación se comunican de forma directa con los objetos concretos de la capa del modelo. Esto reduce la flexibilidad de la arquitectura, ya que el modelo no podría modificarse libremente pues tendría impactos en las interfaces de usuario porque éstas se comunican de una forma prefijada y estática con el modelo. Un cambio en el modelo, podría demandar la modificación no sólo de una sino de varias interfaces de la capa de presentación. Todo este esfuerzo se debe a una falta de flexibilidad, la cual, a su vez, es producto del alto acoplamiento entre la capa de presentación y la capa del modelo.

La arquitectura SOA da una solución a este problema agregando una nueva capa: la Capa de Servicios. Ésta, en la figura anterior, se ubica entre las capas de presentación y la del modelo, tal como se muestra a continuación.

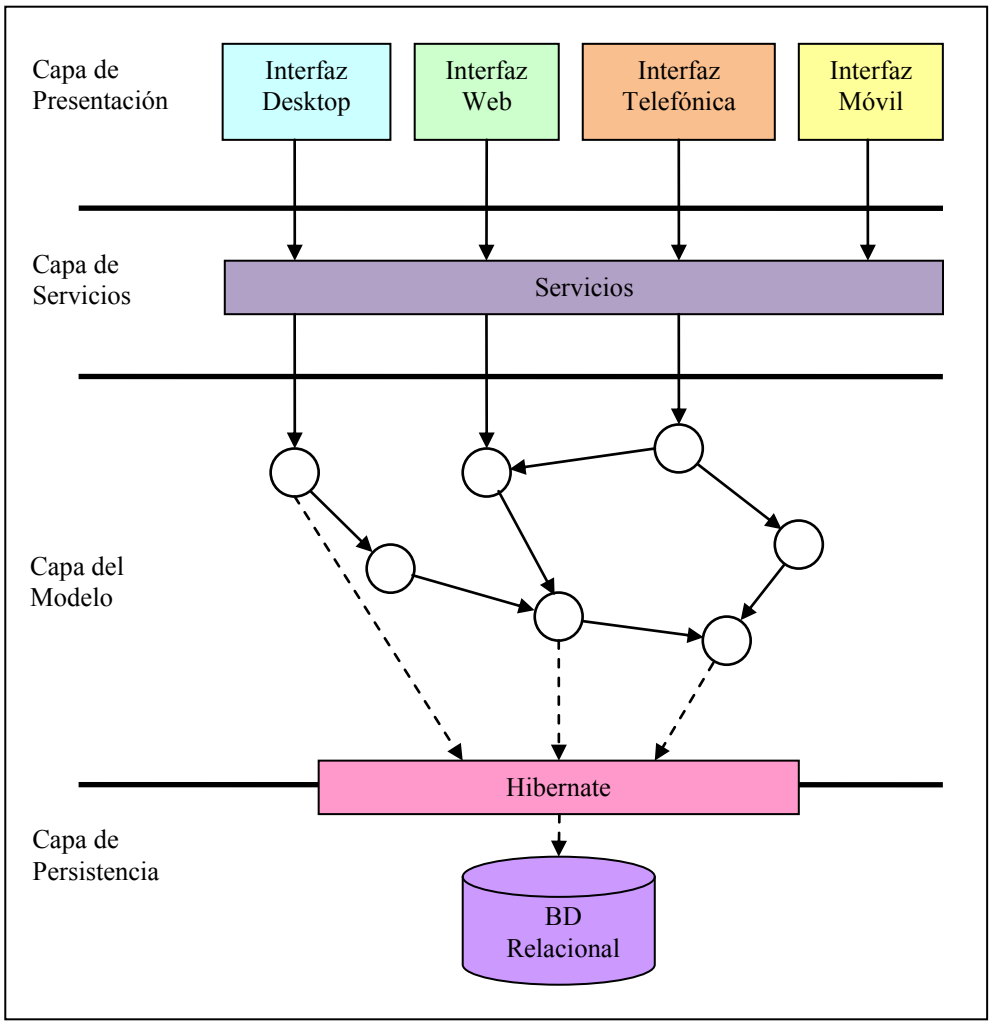


Figura 4.3 – Incorporación de Capa de Servicios

El objetivo de esta capa es "exportar" la funcionalidad que provee el modelo pero ahora en término de servicios. Los servicios son operaciones, que pueden tomar parámetros, y ejecutan la funcionalidad de la capa del modelo enviándole mensajes a sus objetos. De esta forma, ahora las interfaces de la Capa de Presentación se comunican con la Capa de Servicios, invocando la operación que deseen. Al hacerlo, la Capa de Servicios se comunicará con la Capa del Modelo para que éste resuelva el requerimiento y eventualmente pueda devolverse un resultado de la operación.

Como se ve, ahora la Capa de Servicios es la que monopoliza la interacción con el modelo, ya que es ella la única que interactúa con él, a diferencia de la arquitectura previa en donde existía un número grande de interfaces que lo hacían. Como consecuencia de esto se logra mayor flexibilidad ya que, si cambia el modelo, sólo debe cambiarse la forma en que la Capa de Servicios interactúa con el mismo, siendo innecesario, salvo casos puntuales, cambiar la definición de la operación (signature) con lo cual el cambio en el modelo resulta absolutamente transparente para la capa de presentación.

Hasta aquí se ha expuesto la idea de que el único cliente final de la Capa del Modelo es la Capa de Presentación (ahora vía Capa de Servicios). Sin embargo, esta Capa de Servicios no está atada a ninguna forma de presentación particular de la información, por lo que puede ser

utilizada por otros "clientes" que no necesariamente sean interfaces de usuario. En esta línea, estableciendo los permisos adecuados, cualquier otro cliente (por ejemplo, otro módulo de sistema) podría invocar servicios provistos por la Capa de Servicios. Esta interacción de tipo cliente/servidor provista por la capa de servicios facilita enormemente la interacción con otros sistemas que actúen como clientes. Esto es de crucial importancia en nuestro modelo de testing, ya que la aplicación que testeará a esta aplicación podrá ser uno de estos clientes, tal como se explicará en detalle en secciones siguientes.

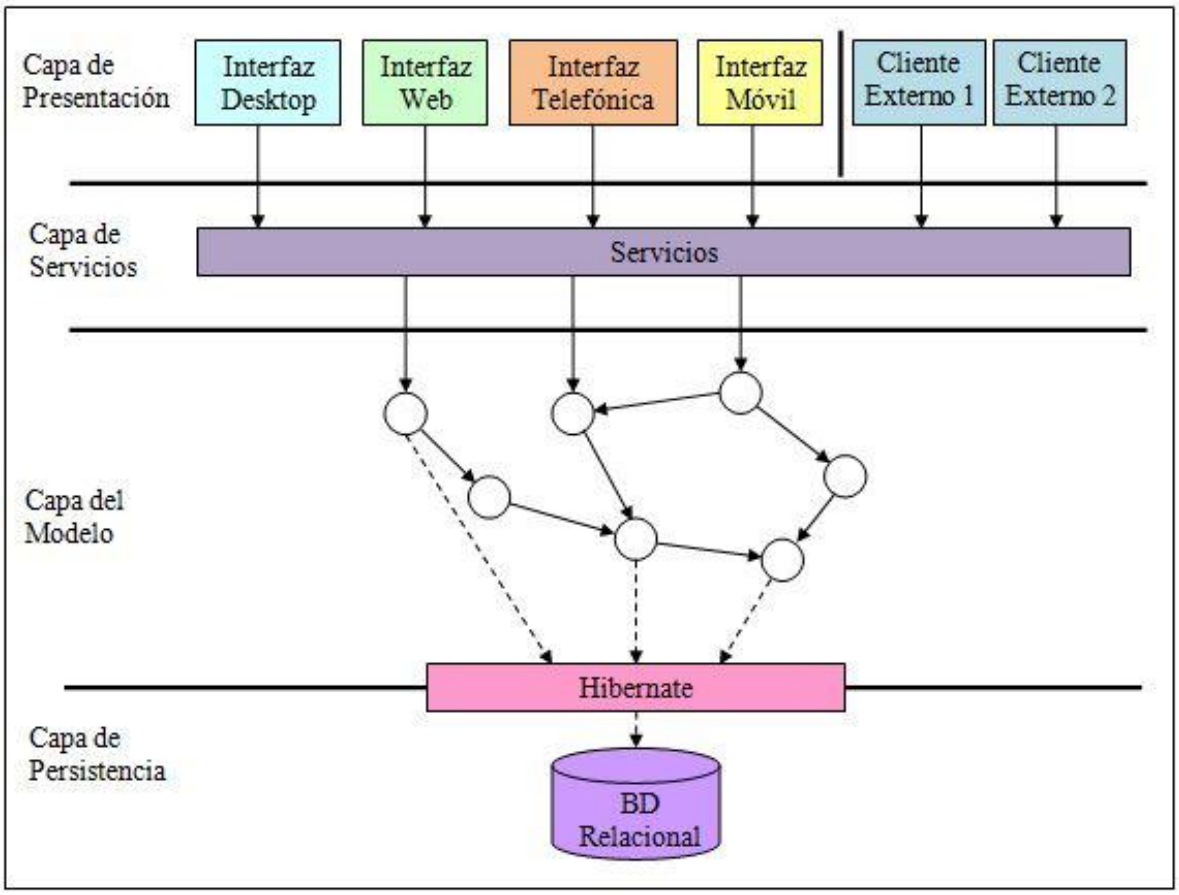


Figura 4.4 – Acceso de clientes externos a la Capa de Servicios

4.1.3. DTOs

Hasta aquí se ha visto cómo se ha logrado desacoplar la Capa de Presentación y la del Modelo en cuanto a las operaciones y los objetos que las implementan. Sin embargo, aún existe acoplamiento en términos de información.

Las operaciones provistas por la capa de servicios pueden tomar parámetros y devolver resultados y, siendo que el modelo está diseñado con objetos, es de esperar que tales parámetros y resultados sean también objetos del modelo a los que, en última instancia, el cliente (por ejemplo, la capa de presentación) terminará conociendo y con los que incluso terminará interactuando. Es decir que el problema es que se están exportando objetos del modelo, con lo cual estamos con la misma naturaleza del problema original: ante modificaciones del modelo que involucren a estos objetos, se verán impactados los clientes que utilicen la Capa de Servicios.

Este problema de acoplamiento, como así también otros de índole transaccional, se puede resolver a partir del Patrón de Diseño DTO (Data Transfer Object). Este patrón propone que

los objetos de la capa del modelo nunca salgan de ella. En vez de ello, propone el uso de objetos que actúen de vehículos para la transferencia de la información entre la capa del modelo y las capas superiores. Estos objetos son los DTO. Un DTO tiene comportamiento prácticamente nulo, y generalmente sólo consta de un conjunto de propiedades: variables de instancia con información y accessors (getters y setters) para acceder y configurar tal información.

En cuanto a la dinámica del uso de los DTOs, cada vez que debe transferirse información de objetos del modelo hacia las capas superiores, la Capa de Servicios copia la la información del objeto del modelo a un objeto DTO y es éste el que envía al cliente. De la misma forma, si el cliente desea modificar la información de este DTO, podrá hacerlo y enviarlo nuevamente a la capa de servicios, quien realizarla la actualización del objeto del modelo con la nueva información del DTO recibido.

Es importante remarcar que la Capa del Modelo no conoce DTOs ya que éstos son objetos que no son parte del dominio de la aplicación, con lo cual la responsabilidad de gestionar la conversión entre DTOs y objetos del modelo es exclusiva responsabilidad de la Capa de Servicios.

Con todo esto se logra un desacoplamiento total entre los clientes y el modelo. La evolución de la arquitectura incluyendo estos DTOs se muestra a continuación.

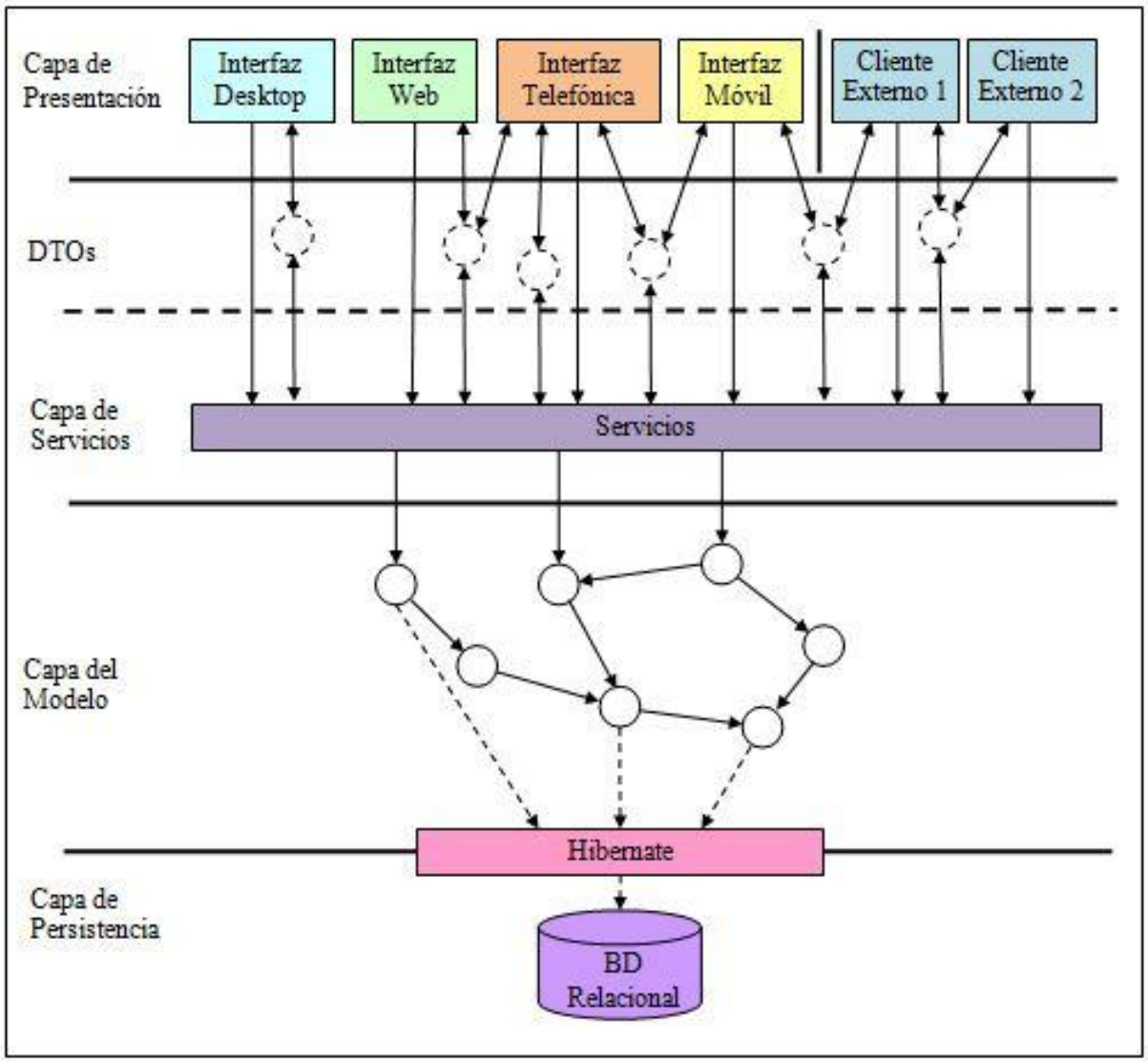


Figura 4.5 – Incorporación de Data Transfer Objects (DTO)

4.1.4. Implementación de los Servicios

En esta sección se describe brevemente el diseño más detallado que permite la implementación de la capa de servicios.

4.1.4.1. Objetos Servicio

Los servicios provistos por la Capa de Servicios se implementan como objetos Java. Cada objeto servicio implementa un conjunto de mensajes que representan un conjunto de operaciones cohesivo. De esta forma, la capa de servicio provee un conjunto de objetos servicio y, cada uno de ellos, provee finalmente un conjunto de mensajes (operaciones) que los clientes pueden invocar.

Con base en lo anterior, un cliente que desee ejecutar una operación de la capa de servicios debe obtener el objeto servicio correspondiente y, a continuación, enviarle a ese objeto el mensaje que representa la operación que desea ejecutar.

4.1.4.2. Localización de Servicios exportados por la aplicación

De acuerdo a lo descripto en la sección anterior, lo primero que debe hacer un cliente que desee ejecutar una operación es obtener una referencia a un objeto servicio. Existen diversas formas de obtener estas referencias, una de ellas es la utilización de locators; en nuestro caso, un ServiceLocator.

El ServiceLocator es un objeto exportado por la aplicación que mantiene referencias a todos los objetos servicio provistos por esta. Este locator provee una interface con un conjunto de mensajes pre-establecido en donde cada uno de ellos devuelve el objeto servicio que corresponda.

En concreto y con base en todo lo anterior, el cliente que desee comunicarse con la aplicación deberá efectuar un procedimiento de dos pasos:

1. Enviar un mensaje al ServiceLocator (exportado por la aplicación) pidiéndole el objeto servicio que contenga la operación que desea ejecutar.
2. Enviar al objeto servicio obtenido el mensaje correspondiente a la operación a ejecutar.

Puede verse que para los clientes el ServiceLocator es el único punto de entrada a aplicación.

4.1.5. Arquitectura final de la aplicación objetivo

Con base en todo lo expuesto hasta aquí, se muestra a continuación la estructura final y detallada de la arquitectura de la aplicación objetivo.

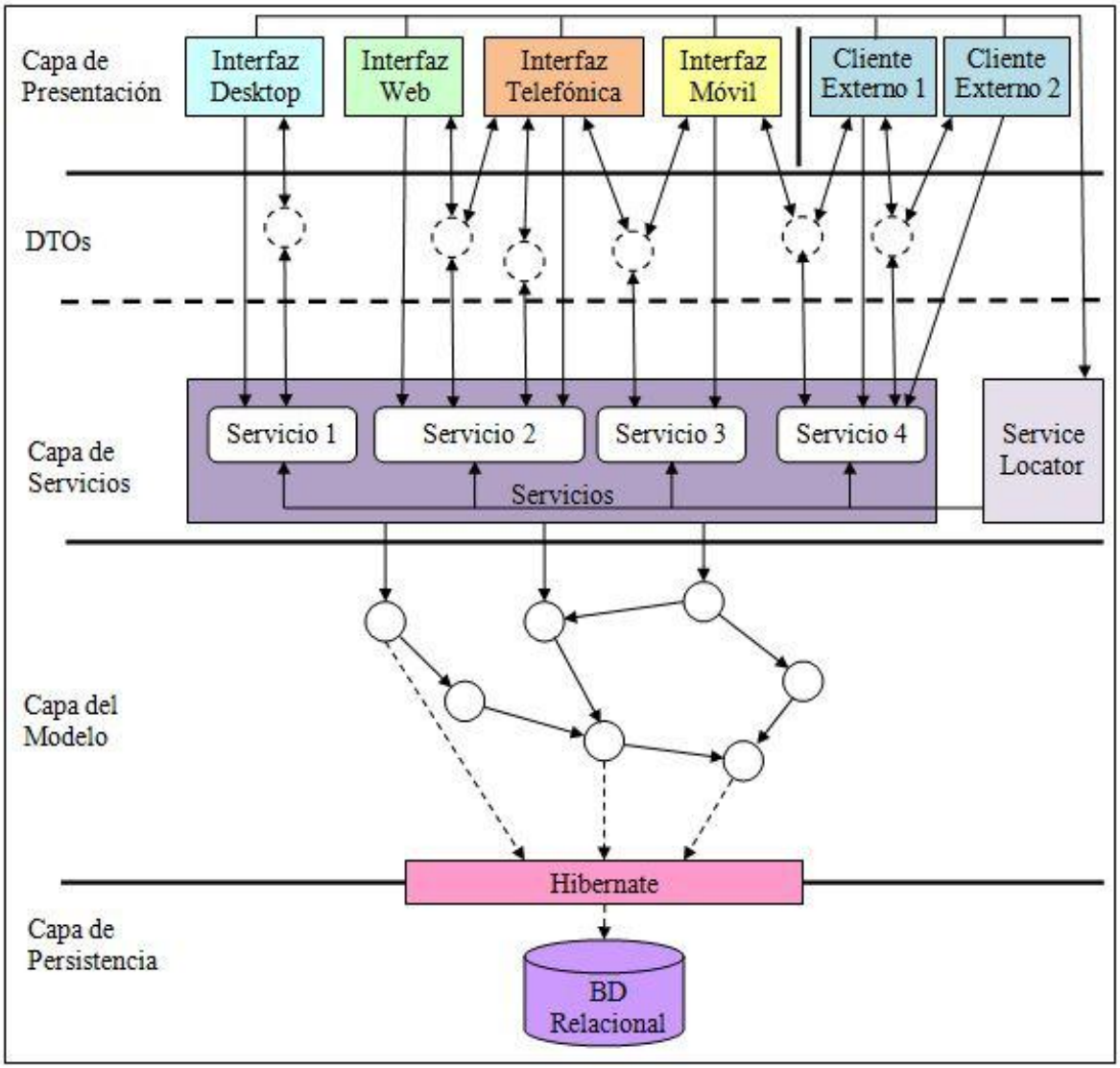


Figura 4.6 – Arquitectura final de la Aplicación Objetivo

4.2. Esquema de Testing para Usuarios Finales

En esta sección se describe el esquema de testing de aplicaciones. Se explican las dos aplicaciones intervinientes, esto es la aplicación a testear y la aplicación utilizada para testearla, como así también la forma en que estas se comunican de forma transparente para lograr generalidad y no invadir la primera de ellas. Asimismo se describen las generalidades de las formas de uso que debe realizar el usuario final.

4.2.1. Definición estática y dinámica de tests

En primer lugar es necesario comprender la diferencia entre la definición estática y la definición dinámica de tests de unidad.

La primera de ellas es la realizada por los programadores que desarrollan tests. Esta definición es realizada directamente mediante código fuente y su foco y entorno de test es bien particular y definido en tiempo de compilación. Es decir que se conocen de antemano y al detalle el escenario de datos, los parámetros, las operaciones a ejecutar y las operaciones de

aserción de resultados. Este tipo de test no puede ser creado ni modificado sin recurrir a la generación y compilación del código y, por lo mismo, debe ser realizada por un profesional con conocimientos de programación.

La segunda forma de definición de tests, la dinámica, implica que puede ser realizada en tiempo de ejecución (ya no en tiempo de compilación), por lo que toda la información de datos, parámetros y operaciones no es conocida de antemano. Esto significa que deben proveerse mecanismos lo suficientemente genéricos y flexibles para adaptarse a una diversidad de situaciones que se especificarán en un momento posterior a la compilación, en el cual podrían programarse los tests de forma directa y en su máximo nivel de detalle de acuerdo a los contextos de tests particulares, tal como se describió en el párrafo anterior. Esta segunda forma permitiría que, mediante las herramientas adecuadas, usuarios finales puedan crear y configurar tests de unidad, lo cual constituye el objetivo de este trabajo.

4.2.2. Aplicación Objetivo y Aplicación de Testing

Las dos componentes básicas del esquema son la Aplicación Objetivo y la Aplicación de Testing.

La primera es la aplicación a testear, la que se constituye como objetivo del test. Esta aplicación tiene la arquitectura descripta en la sección anterior, de amplio uso en la industria, pero no se exigen adaptaciones particulares a la misma para poder ser testeada. Justamente esta generalidad constituye uno de los lineamientos de diseño perseguidos a lo largo de este trabajo.

Por otro lado, la aplicación de testing, es la herramienta que utilizará el usuario final (no experto en temas técnicos) para definir y configurar el conjunto de tests que se ejecutarán sobre la aplicación objetivo. Estos tests serán creados en tiempo de ejecución y por un usuario no experto (ya no programadoes), por lo que tales definiciones deben ser dinámicas y mediante interfaces gráficas de gran usabilidad.

De esta manera, la aplicación de testing se conecta a la aplicación objetivo, sin que esta tenga conocimiento alguno de aquella, logrando transparencia y no intrusión.

A continuación se muestra un diagrama sencillo con estas dos componentes, el cual se irá evolucionando en secciones y capítulos posteriores.

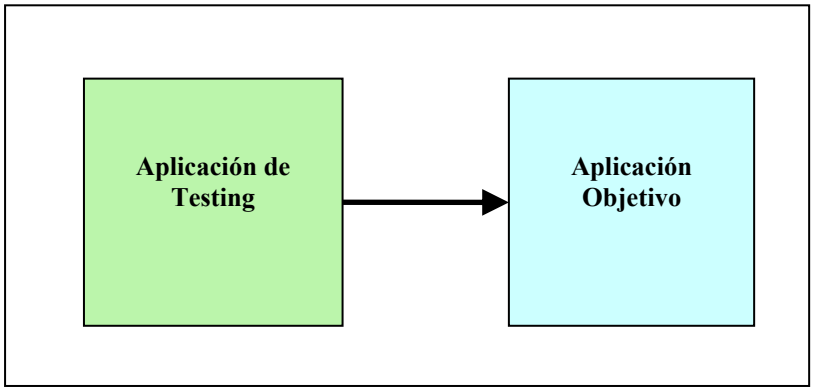


Figura 4.7 – Esquema básico de Testing

4.2.3. Conexión de las dos aplicaciones

Tal como se explicó en secciones anteriores, la aplicación objetivo provee una capa de servicios que puede ser utilizada por distintos clientes. En este contexto, la aplicación de testing se constituye como cliente de la misma, realizando la ejecución y aserción de tests mediante las operaciones provista por esa capa de servicios.

Asimismo, y como se ha descripto, la aplicación de testing contará con el ServiceLocator que exporta la aplicación objetivo, de forma que la primera pueda contar un punto de acceso a los objetos servicio que necesite utilizar.

A continuación se evoluciona la figura anterior, mostrando esta conexión a nivel servicios.

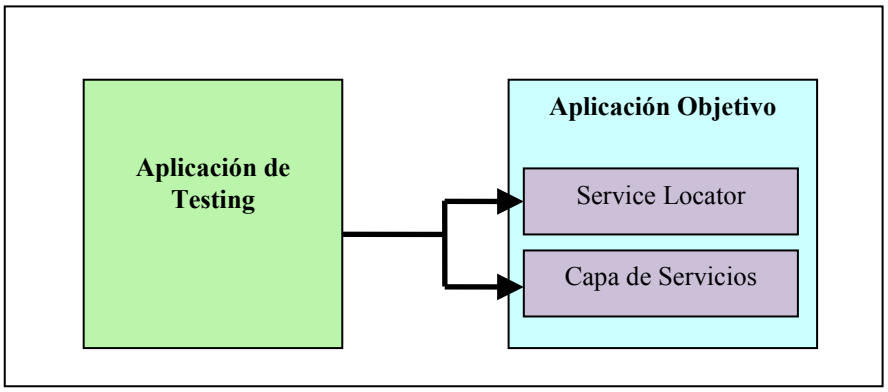


Figura 4.8 – Forma de conexión a la Aplicación Objetivo

4.2.4. Exportación de Servicios

Hasta aquí, en el contexto de la ejecución de tests la aplicación de testing puede invocar servicios de la aplicación objetivo, por lo cual a nivel técnico este problema se encuentra resuelto. Sin embargo, el usuario final de la aplicación de testing debe poder saber qué operaciones puede ejecutar, qué información debe proveer para tales operaciones, qué resultados devuelven esas operaciones, y todas las descripciones y explicaciones referentes a estos aspectos. Para lograr esto se deben proveer herramientas visuales de alta usabilidad de forma tal que el usuario pueda crear y configurar los tests de manera amena.

El elemento básico para conseguir esta funcionalidad es el Descriptor. Un descriptor es un elemento que permite manipular de forma indirecta y amena a otro elemento de índole técnica y por lo tanto intratable de forma directa por un usuario final. En este sentido, existen descriptores para varios elementos técnicos provistos por la aplicación objetivo, como ser servicios, operaciones, parámetros y resultados entre otras cosas.

Un Descriptor provee dos tipos de información:

- Información Técnica: es información invisible al usuario final, pero que permite al sistema establecer la asociación entre el descriptor que manipula el usuario y el elemento técnico del modelo que representa. Además, esta información técnica es utilizada internamente por las interfaces de usuario, ya que basándose en ella pueden implementar soluciones ágiles y dinámicas de cara al usuario final.

- Información Descriptiva: es la información visible para el usuario final, y apunta a facilitarle la gestión de los elementos técnicos, ocultándoselos. Esta información incluye nombres y descripciones acordes al dominio del usuario. De esta forma, el usuario se desliga de los nombres técnicos de, por ejemplo, mensajes y parámetros de Java, teniendo a cambio nombres que le resultan propios a su dominio. De la misma forma, se agregan descripciones a estos elementos para orientarlo y mantenerlo en control de la situación.

De esta forma, los descriptores permiten al usuario final de la aplicación de testing manipular de forma amena a los servicios provistos por la aplicación objetivo.

Cada descriptor puede ser representado mediante una estructura XML, por lo cual varios de ellos pueden ser incluídos en un archivo de este tipo. Es común que un mismo elemento puede ser representado de dos maneras posibles: como objeto viviente dentro de un modelo de aplicación o como XML dentro de un archivo. Es también práctica común que pasar de una representación a otra y viceversa. Estas transformaciones de representación se realizan típicamente en los procesos de exportación e importación, y esto es justamente lo que se realiza en este trabajo.

Con base en todo lo anterior, los servicios provistos por la aplicación objetivo pueden ser representados mediante objetos Descriptores, y éstos, a su vez, transformados a representación XML e incluidos en un archivo; en esto consiste el proceso de exportación.

Los elementos de este archivo pueden editarse de forma tal de enriquecerlos con toda la información descriptiva que sea útil al usuario final de la aplicación de testing. Esto puede realizarse tanto a mano como con alguna herramienta de edición concebida para este fin.

Posteriormente, puede realizarse el proceso de importación de este archivo XML enriquecido, lo cual consiste en la transformación de los descriptores del archivo en objetos, y en la incorporación de esos objetos descriptores en la aplicación de testing.

Con todo esto, el usuario final puede manipular elementos técnicos de forma amena, pudiendo así crear y configurar tests.

En resumen, la manipulación de servicios de la aplicación objetivo desde la aplicación de testing se realiza mediante el siguiente proceso:

1. Exportación de la información técnica de los servicios de la aplicación objetivo, en forma de descriptores, a un archivo en formato XML.
2. Enriquecimiento de tales descriptores del archivo con información descriptiva, útil al usuario final.
3. Importación de los descriptores del archivo XML del punto anterior a la aplicación de testing.

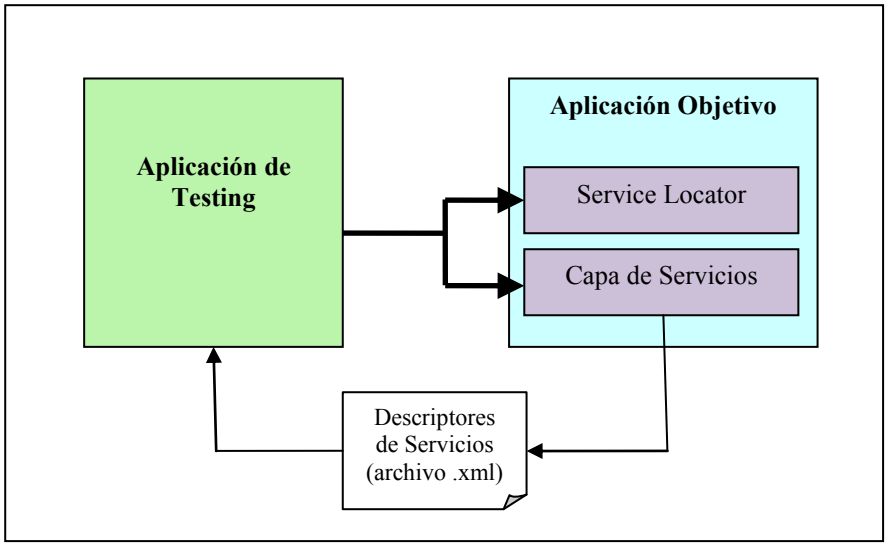


Figura 4.9 – Exportación e Importación de Servicios de la Aplicación Objetivo

4.2.5. Uso básico de la aplicación de testing

Si bien más adelante se destina una sección completa a la explicación del uso de la aplicación de testing, aquí se brinda una muy breve descripción de tal uso. Esta explicación tiene el objetivo de brindar al lector un contexto general acabado del esquema general, tanto

estructural como dinámico.

El esquema de uso general es el siguiente:

1. Exportación, enriquecimiento e importación de los descriptores de servicios de la aplicación objetivo, tal como se describió en la sección anterior. Esto debe hacerse una única vez, o cada vez que se desee agregar o modificar un servicio.
2. Creación y configuración de tests mediante los descriptores importados en el punto anterior. Esto incluye también las operaciones de aserción de resultados y la especificación del juego de datos de base a utilizar.
3. Ejecución de los tests creados.

El segundo punto implica la definición, para cada test, de un juego de datos sobre los que se ejecutará el mismo. La forma de creación de este juego de datos es provista por DBUnit y consiste en la exportación de toda o un área de la base de datos subyacente de la aplicación objetivo a un archivo XML. El usuario final, que puede operar la aplicación objetivo, podrá con ella misma conformar el juego de datos que desee y, una vez hecho esto, disparar la exportación al archivo mencionado, para finalmente adjuntarlo al test que está creando.

El tercer punto consiste en el funcionamiento básico de DBUnit, pero ahora aplicando todo un conjunto de técnicas reflexivas se Java que posibilitan el dinamismo necesario para permitir la creación y configuración por parte de un usuario final. Tal como se mencionó en secciones anteriores, la ejecución de un test consiste en los siguientes pasos:

- Carga de la base de datos de la aplicación objetivo con la información del juego de datos del test que se encuentra en el archivo asignado al mismo.
- Ejecución sobre la aplicación objetivo de las operaciones del test, las cuales modifican el estado del modelo y la base de datos de la aplicación objetivo.
- Ejecución sobre la aplicación objetivo de las operaciones de aserción de resultados esperados.

Como resultado de esto, la aplicación de testing almacena los resultados correspondientes de los tests haciéndolos visibles al usuario de la misma.

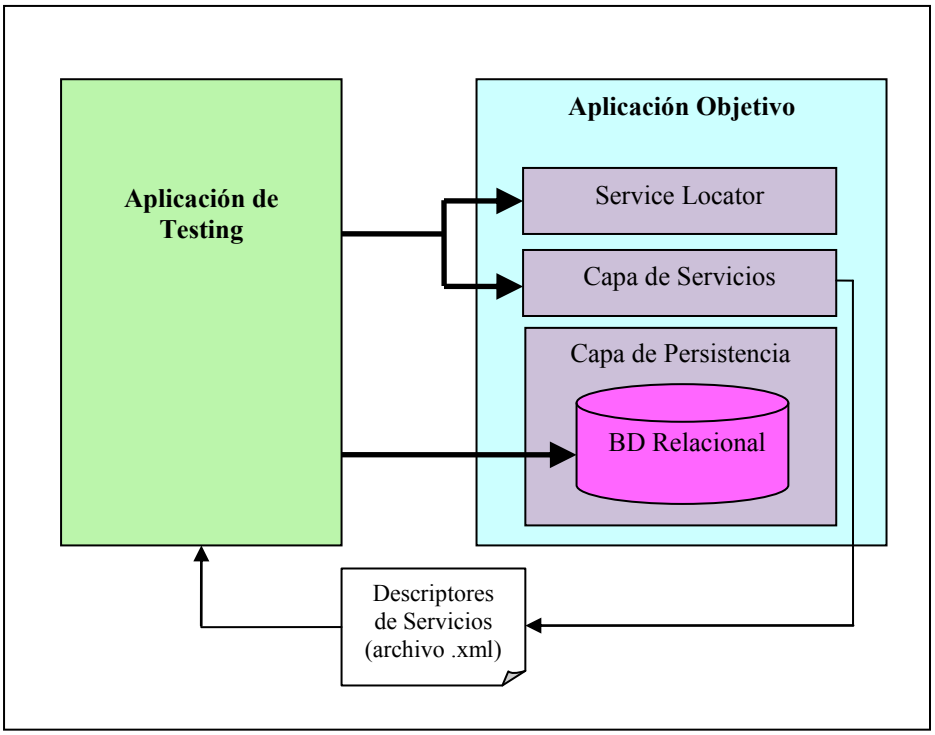


Figura 4.10 – Setup del juego de datos persistente durante un test

4.2.6. Persistencia de la aplicación de testing

Si bien la persistencia de la aplicación objetivo se realiza mediante las técnicas y tecnologías ya descriptas, pues son las que la aplicación de testing espera encontrar, no hay limitaciones en cuanto las formas de persistencia de la aplicación de testing.

Esta última aplicación está diseñada de forma que pueda ser persistida o no; de hecho puede ser utilizada sin ningún tipo de persistencia o bien persistida mediante otras tecnologías. Esto aporta un grado más de flexibilidad que, si bien no se relaciona con la flexibilidad necesaria para el dinamismo en la ceración y ejecución de tests, permite mayor libertad de uso, configuración e instalación.

5. MODELO BÁSICO DE LA APLICACIÓN DE TESTING

En esta sección se desarrolla el modelo básico de la Aplicación de Testing. Este modelo permite realizar tests para una amplia variedad de casos, entre ellos, y mediante una extensión que considera ciertas particularidades, el de aplicaciones con una arquitectura como la ya descripta para la Aplicación Objetivo.

5.1. Lineamientos de Diseño

Los siguientes son algunos lineamientos de diseño que se siguen a lo largo de todo el desarrollo del modelo:

- La Aplicación de Testing podrá ser persistida o no, y de serlo, la tecnología utilizada debe poder ser seleccionada libremente.

- El Modelo Básico que se describe a continuación debe presentar una alta flexibilidad y adaptabilidad a una gran variedad de usos. El uso del modelo básico con servicios en arquitecturas SOA (como la de la Aplicación Objetivo descripta), se constituye sólo como un caso particular y de ninguna manera limitativo.

- Ninguna decisión de diseño debe ser intrusiva en relación a la Aplicación Objetivo. Se entiende que ésta es construida para sus propios fines, y que en esa construcción de ninguna manera se deben contemplar aspectos particulares de su testing posterior a través de la Aplicación de Testing.

En capítulos posteriores se extenderá este Modelo Básico para adaptarlo a un contexto más acotado según elecciones y necesidades, como por ejemplo su persistencia mediante Hibernate y su uso para el testing de Aplicaciones Objetivo de tipo SOA implementadas en Java.

5.2. Dinamismo mediante Reflection

En capítulos anteriores se describieron los conceptos de Definición Estática y Definición Dinámica de tests. Una de las técnicas más utilizadas en este trabajo para lograr este último tipo de definición es Reflection.

El modelo de dominio del problema de una aplicación consta de clases, objetos y mensajes que abstraen aspectos particulares de ese dominio. Así, en un sistema de información financiera, tendremos cuentas bancarias, clientes y operaciones de depósito, extracción y transferencia, entre otras. Toda la operatoria e información necesaria puede realizarse y obtenerse a partir de la interacción de esos objetos ejecutando los mensajes correspondientes. Tal es el caso de la Aplicación Objetivo.

No obstante esto, existen otras aplicaciones para las cuales el dominio de aplicación está constituído por el mismo paradigma de orientación a objetos, por lo que los elementos con que trabajan serán clases, objetos y métodos, pero ya no representando para ellas algo de un dominio particular, sino lo que en sí ya representan en el paradigma. Estas aplicaciones deberán trabajar en tiempo de ejecución con esos elementos (instanciar clases, crear objetos e invocar mensajes, sin conocerlos de antemano en tiempo de compilación). Tal es el caso de la Aplicación de Testing, que deberá interactuar con clases y objetos de la Aplicación Objetivo de forma genérica, independientemente del dominio de aplicación al que pertenezcan. La forma en que se descubren y obtienen esas clases y objetos desconocidos, como así también la forma en que se los hace interactuar mediante la ejecución de mensajes (también desconocidos

de antemano), es Reflection.

El trabajo con Reflection implica trabajar en un nivel de abstracción mayor y, debido a esto, el tipo de aplicaciones descripta en este trabajo requiere realizar algunas acciones nada usuales, como por ejemplo almacenar un método en una base de datos. Este tipo de operaciones es el que requiere el mayor esfuerzo, pero a su vez el que permite lograr el dinamismo propio de la naturaleza de este trabajo.

5.3. Diseño del Modelo Básico

En esta sección se realiza la descripción gradual del modelo básico de la Aplicación de Testing, a la vez que se discuten distintas alternativas de diseño y sus justificaciones.

5.3.1. Test Dinámicos: extensión de Frameworks preexistentes.

El modelo básico extiende la funcionalidad existente de los frameworks JUnit y DBUnit, agregándoles capacidad dinámica de creación, configuración y ejecución. Estas características son justamente las que permiten realizar las gestiones de tests sin necesidad de programación y compilación, lo que a su vez permite prescindir del programador y delegar estas tareas a un usuario final.

En la siguiente figura se muestra esta extensión de frameworks, la cual se realiza mediante herencia.

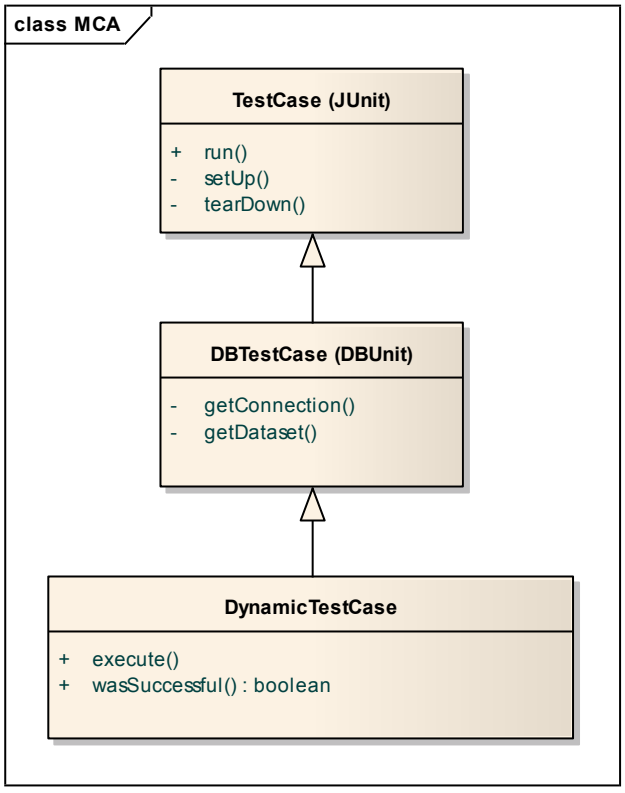


Figura 5.1 – Extensión de Frameworks existentes

La jerarquía de la figura anterior, permite que cada clase agregue nueva funcionalidad a la anterior, de la siguiente manera:

- TestCase: pertenece al framework JUnit y representa a un Test de Unidad en Java. Implementa la funcionalidad relacionada a la creación de tests, gestión de escenarios de datos (setup y tier-down), ejecución de tests, aserción de resultados obtenidos y gestión de test suites.

- DBTestCase: pertenece al framework DBUnit y representa a un Test de Unidad en Java al igual que JUnit, pero posibilitando, además, la ejecución de tests sobre ambientes persistentes. Esto significa que los escenarios de datos ya no necesitan ser creados explícitamente en memoria por el programador, sino que pueden ser tomados de la Base de Datos de la aplicación a testear, de la misma forma que los resultados del test también podrán ser verificados contra la misma fuente de información de la aplicación.

- DynamicTestCase: es la clase principal de test creada para este trabajo, y extiende la funcionalidad de testing general y testing sobre ambientes persistentes (provistas por las dos superclases mencionadas) agregándoles dinamismo. Esto significa básicamente que no se necesitará de un programador para la creación, configuración y ejecución de tests, sino que todas estas gestiones podrán ser realizadas por un usuario final y en tiempo de ejecución.

5.3.2. Modelo de ejecución de operaciones

Tal como se ha descripto en los primeros capítulos, luego de cargar un juego de datos un test ejecuta un conjunto de operaciones sobre el mismo y, posteriormente, ejecuta un conjunto de operaciones de aserción de forma de comparar los resultados obtenidos contra los resultados esperados. Llamaremos “Operaciones de Ejecución” a las primeras, mientras que a las segundas las llamaremos “Operaciones de Aserción”.

Antes de examinar cómo estas operaciones se relacionan con el modelo de testing, se describirá una abstracción muy útil a tales efectos: la operación. Ésta abstracción encapsula, entre otras cosas, la configuración de una operación, su forma de ejecución y el resultado de la misma. En concreto, al ejecutar una operación se envía un mensaje a un objeto, tal mensaje define un conjunto de parámetros y la ejecución del mismo devuelve un resultado.

En el modelo básico, este encapsulamiento se obtiene mediante la aplicación del patrón Command, dando lugar a la clase OperationCommand. Se podría hablar de dos momentos esenciales por los que atraviesan las instancias de esta clase: configuración y ejecución. En la primera, a la instancia se le especifican todos objetos concretos que necesitará posteriormente durante la ejecución. Con esto se logran instancias auto-contenidas de forma tal que en el segundo momento, el de ejecución, sólo debe indicárseles que se ejecuten, sin necesidad de especificar ningún tipo de parámetros ni contexto, e incluso sin saber concretamente qué es lo que harán. En este sentido, es de destacar el alto grado de desacoplamiento e independencia que los clientes de estas instancias logran con respecto a ellas en el momento de la ejecución.

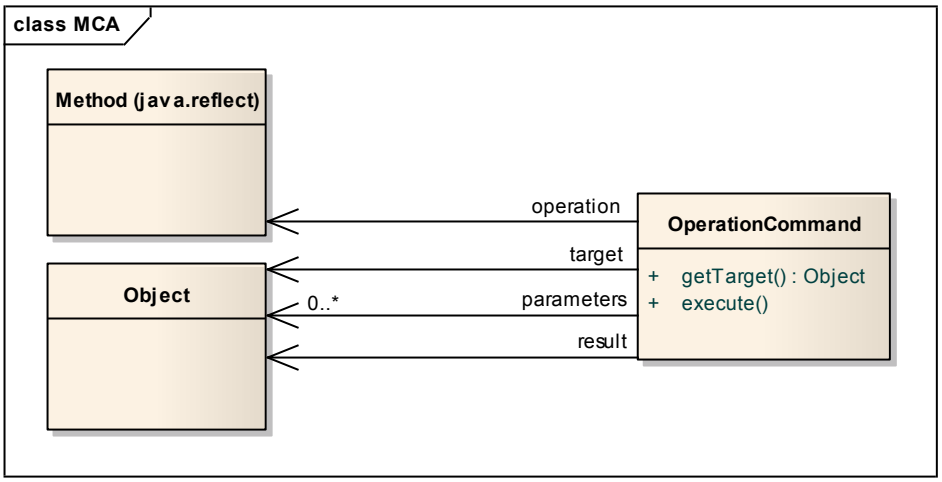


Figura 5.2 – Modelo genérico de ejecución de operaciones

En la figura anterior puede verse que la clase OperationCommand tiene cuatro elementos configurables que son básicos:

- Target: es el objeto concreto que ejecutará la operación definida.
- Operation: es el mensaje que se enviará al target, lo cual disparará la ejecución de la operación.
- Parameters: son los parámetros que tendrá el mensaje al ser enviado al target.
- Result: es el resultado obtenido del envío del mensaje (con los parámetros definidos) al target.

Aquí ya puede observarse el alto grado de flexibilidad que debe proveer el modelo, ya que estos objetos (target, parameters y result) podrían ser de cualquier clase, mientras que el mensaje también podría ser cualquiera que el objeto target implemente.

En la creación estándar de tests mediante JUnit o DBUnit, estos targets, mensajes y parámetros, son conocidos en detalle por el programador, quien crea y compila un programa específico para ese contexto particular. Por el contrario, en nuestro modelo todos estos detalles no son conocidos en tiempo de compilación, pero deben poder ser abarcados por la configuración del usuario final en momento de ejecución. En el primer caso hablamos de un contexto "estático", mientras que el segundo caso hablamos de un "contexto dinámico". En el modelo básico, este dinamismo se logra mediante reflection, y particularmente en el caso del OperationCommand se logra mediante las clases Object y Method.

Dado que constituye un punto central en este trabajo, a continuación se explicitará la diferencia entre la definición estática de una operación, tal como la escribiría un programador en tiempo de compilación, y su contraparte dinámica, tal como la podría definir un usuario final en tiempo de ejecución mediante el modelo aquí propuesto. Para visibilizar más claramente esta diferencia, se introduce un pequeño ejemplo concreto.

Este ejemplo consiste en una Compañía Telefónica, la cual gestiona un conjunto de Líneas Telefónicas que pertenecen a los abonados que contratan sus servicios. En determinados períodos esta compañía emite Facturas (bills) para cada línea teléfonica, de forma tal que pueda cobrar a sus abonados por el total de llamadas realizadas desde esa línea en ese período.

A continuación se muestra el diagrama de clases del modelo del dominio.

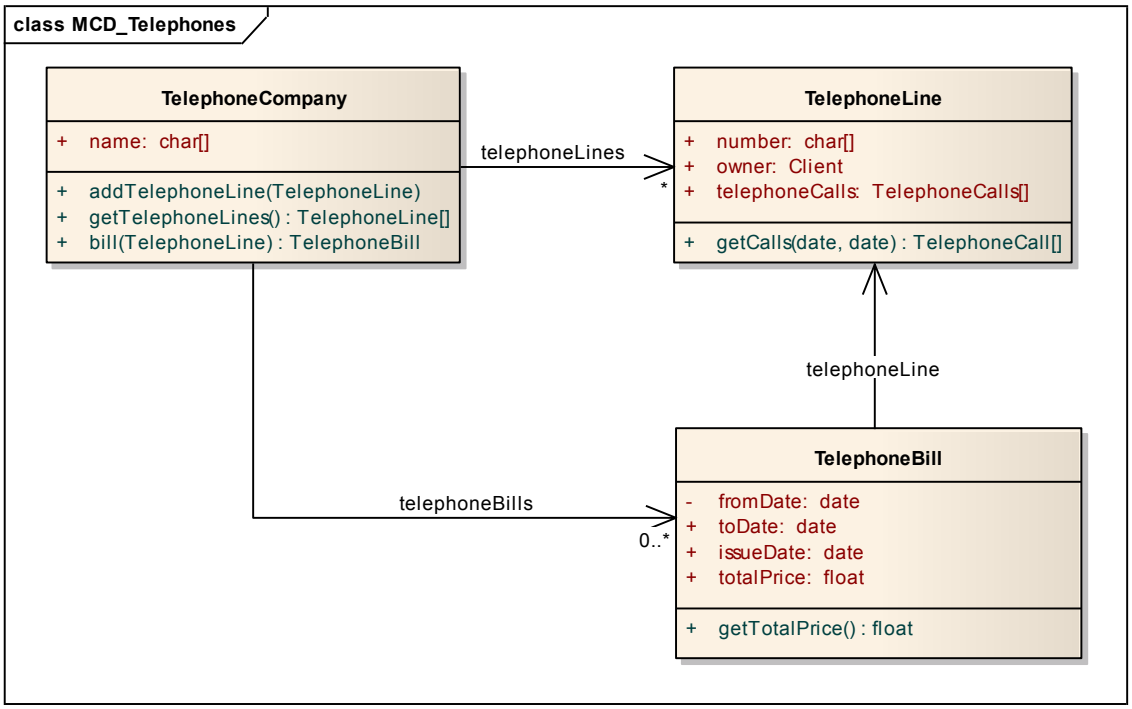


Figura 5.3 – Modelo de clases de la aplicación telefónica.

A continuación se muestra un diagrama de objetos que representa una instanciación concreta del modelo de clases anterior.

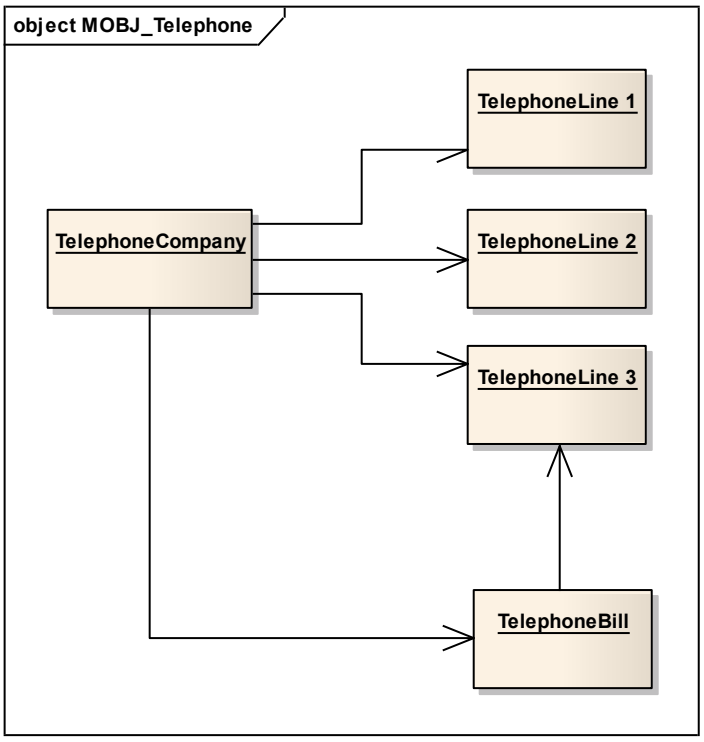


Figura 5.4 – Modelo de Objetos de una instanciación concreta

A partir de este modelo y caso puntual de instanciación, la forma estática de ejecución de una operación (en particular, la generación de una factura) se muestra a continuación:

```
TelephoneCompany telephoneCompany =  <inicialización>.
TelephoneLine telephoneLine3:= telephoneCompany.getLines().last().
TelephoneBill telephoneBill = telephoneCompany.bill(telephoneLine3);
```

Figura 5.5 –Ejecución Estática de una Operación

En la siguiente figura se muestra la forma en que la misma operación se configura para su ejecución dinámica mediante un OperationCommand.

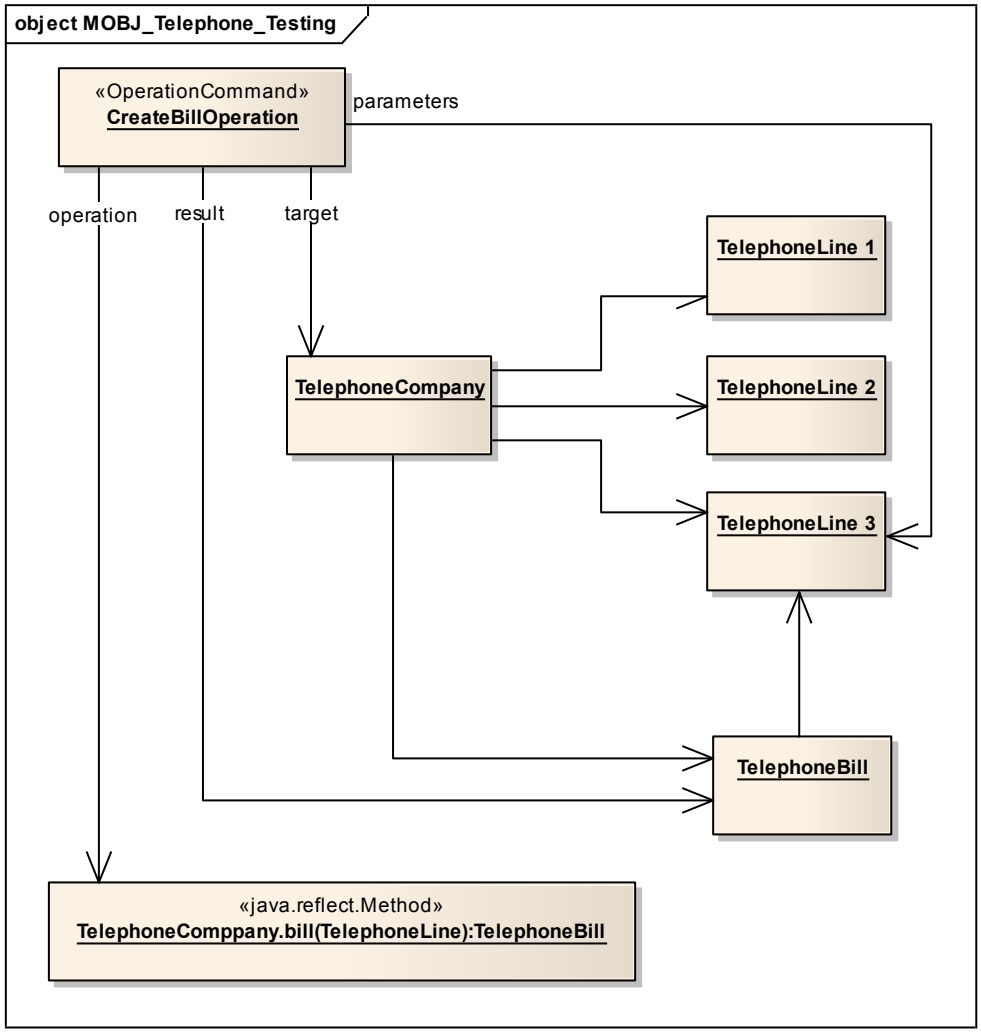


Figura 5.6 - Ejecución Dinámica de una Operación

La ejecución de la operación se dispara mediante el envío del mensaje execute() al OperationCommand. En ese momento éste ejecutará, mediante reflection, el método que conoce como su operation (en este caso, bill(TelephoneLine)) con los objetos parámetro que conoce a través de su variable parameters (en este caso, el objeto TelephoneLine3) sobre el objeto que conoce como su target (en este caso, el objeto TelephoneCompany). El resultado de

esta ejecución devuelve un objeto (que en este caso es una TelephoneBill), y setea como su resultado, result.

Este mismo proceso se muestra a continuación mediante un diagrama de secuencia.

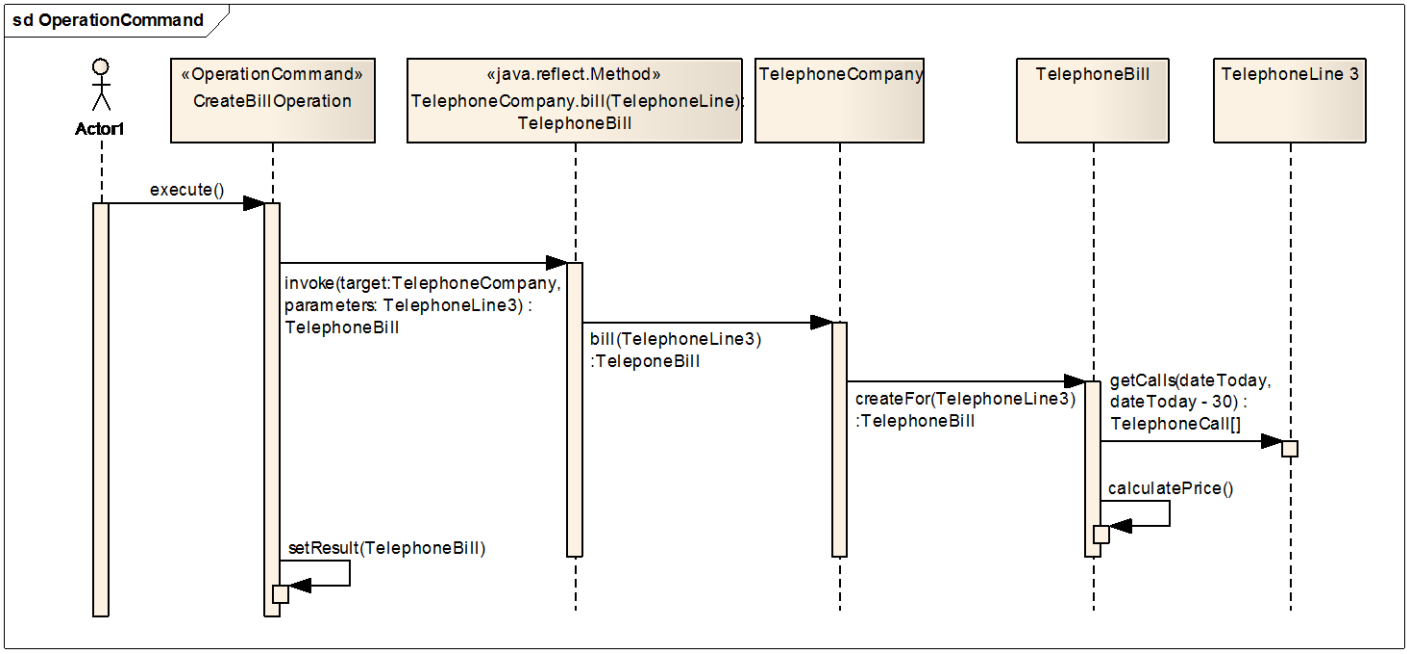


Figura 5.7 – Proceso de Ejecución Dinámica de una Operación

El modelo del diagrama anterior cubre las necesidades cuando el resultado final que se busca es exactamente el objeto devuelto por la operación ejecutada. Sin embargo, frecuentemente, el resultado que se busca es en realidad un objeto conocido por el objeto devuelto. Esto implica que para obtener ese segundo objeto se debe enviar un mensaje al primer objeto, es decir al objeto devuelto por la operación original. Esto podría extenderse y encadenarse de forma tal que a ese segundo objeto pueda enviársele otro mensaje, obteniendo un tercer objeto, y así sucesivamente conforme sea la necesidad de lograr mayor profundidad de navegación a partir del objeto resultado original.

En el modelo esta posibilidad se logra definiendo una colección de mensajes "resultAccessorOperations" en el OperationCommand. De esta forma, al ejecutar el OperationCommand, éste inmediatamente enviará en cascada esta colección de mensajes, el primero de ellos al objeto obtenido de la ejecución de la operación original, y los restantes al resultado obtenido del envío del mensaje anterior en la cadena.

Otro aspecto a tener cuenta es que, en determinados contextos, el objeto que debe ejecutar la operación original (target) no está disponible en todo momento sino que debe ser obtenido de alguna fuente externa, ya sea en el momento de inicio de la aplicación o en el momento preciso de la ejecución de la operación. Esto puede suceder, por ejemplo, con objetos remotos o con objetos versionables, es decir en aquellos casos en que se debe actualizar la versión del objeto.

En el modelo básico esto se resuelve mediante la clase TargetLocator, a quien el OperationCommand le delega la responsabilidad de localizar y traer al contexto de ejecución al objeto target que utilizará. La forma concreta en que el TargetLocator realiza esta búsqueda es transparente al OperationCommnad, limitándose éste sólo a pedírselo. El diseño queda abierto de forma tal que puedan crearse nuevos tipos de TargetLocators según distintas necesidades, ya que el OperationCommand interactúa con tales instancias de manera polimórfica mediante el envío de un único mensaje: getTarget().

A continuación se muestra el Diagrama de Clases de esta parte del modelo, extendido con estas dos funcionalidades (operaciones de acceso al resultado y localización del target) y la extensión del ejemplo dado previamente para ilustrar su instanciación.

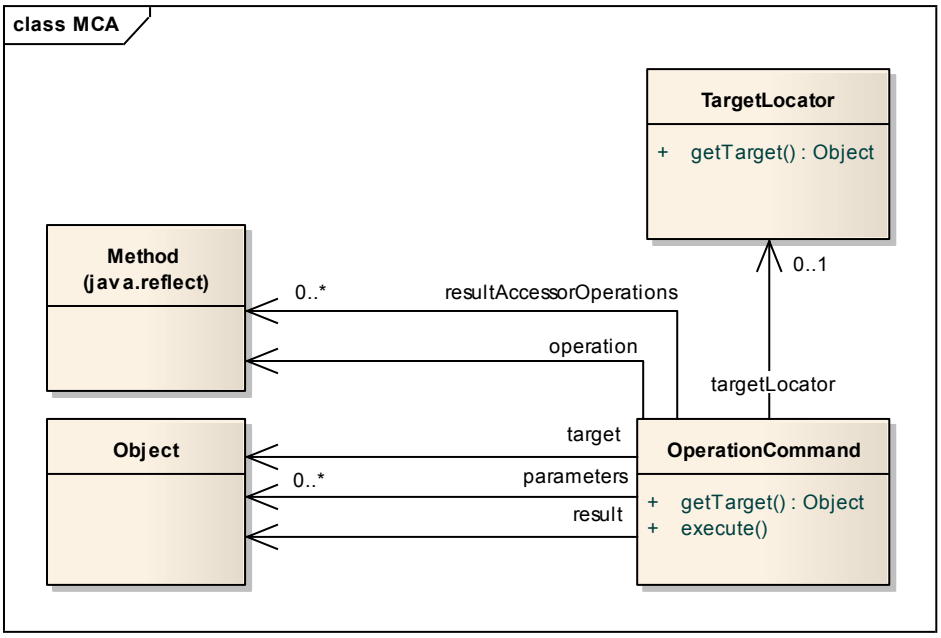


Figura 5.8 – Target Locator y Operaciones de Acceso al Resultado

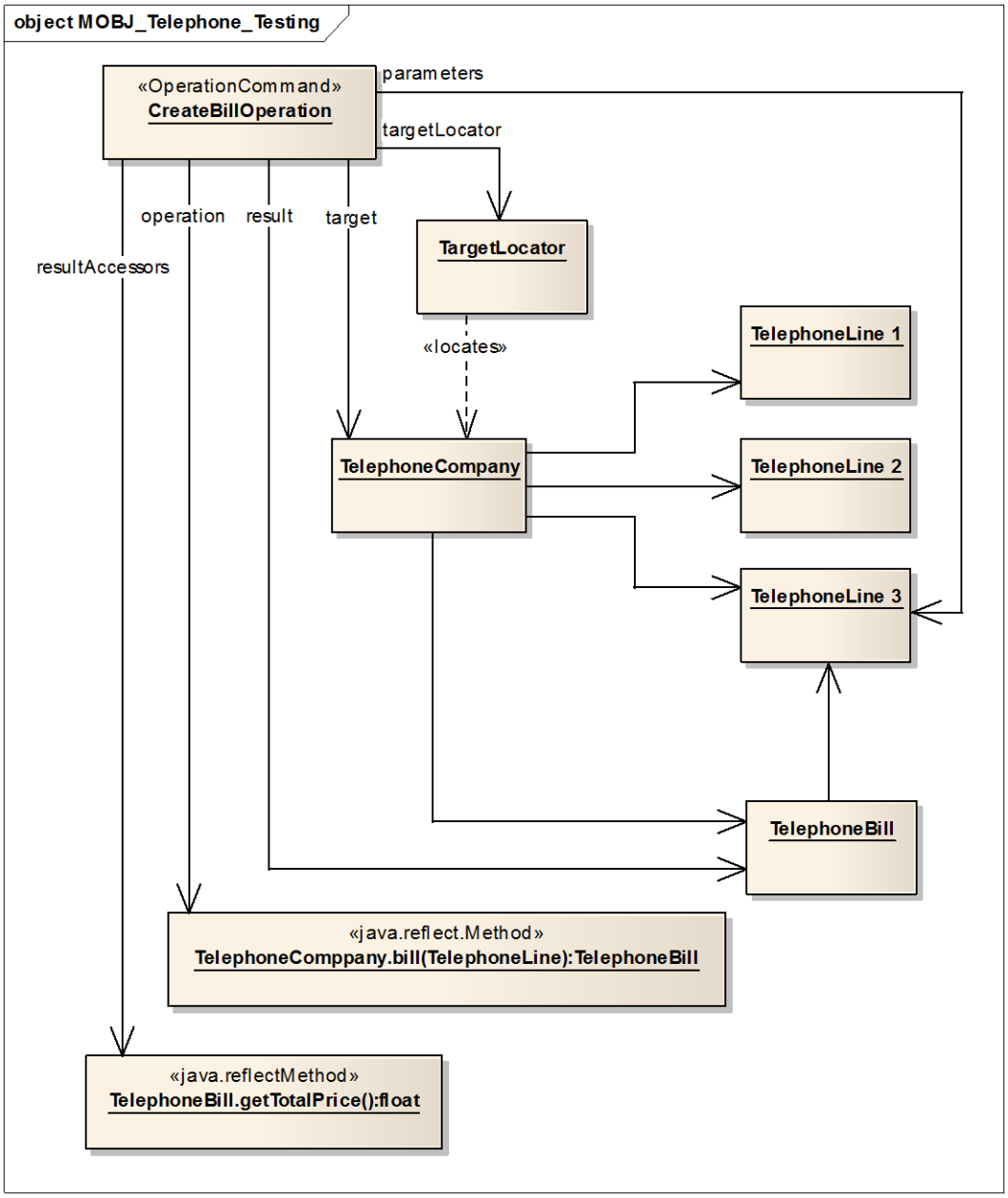
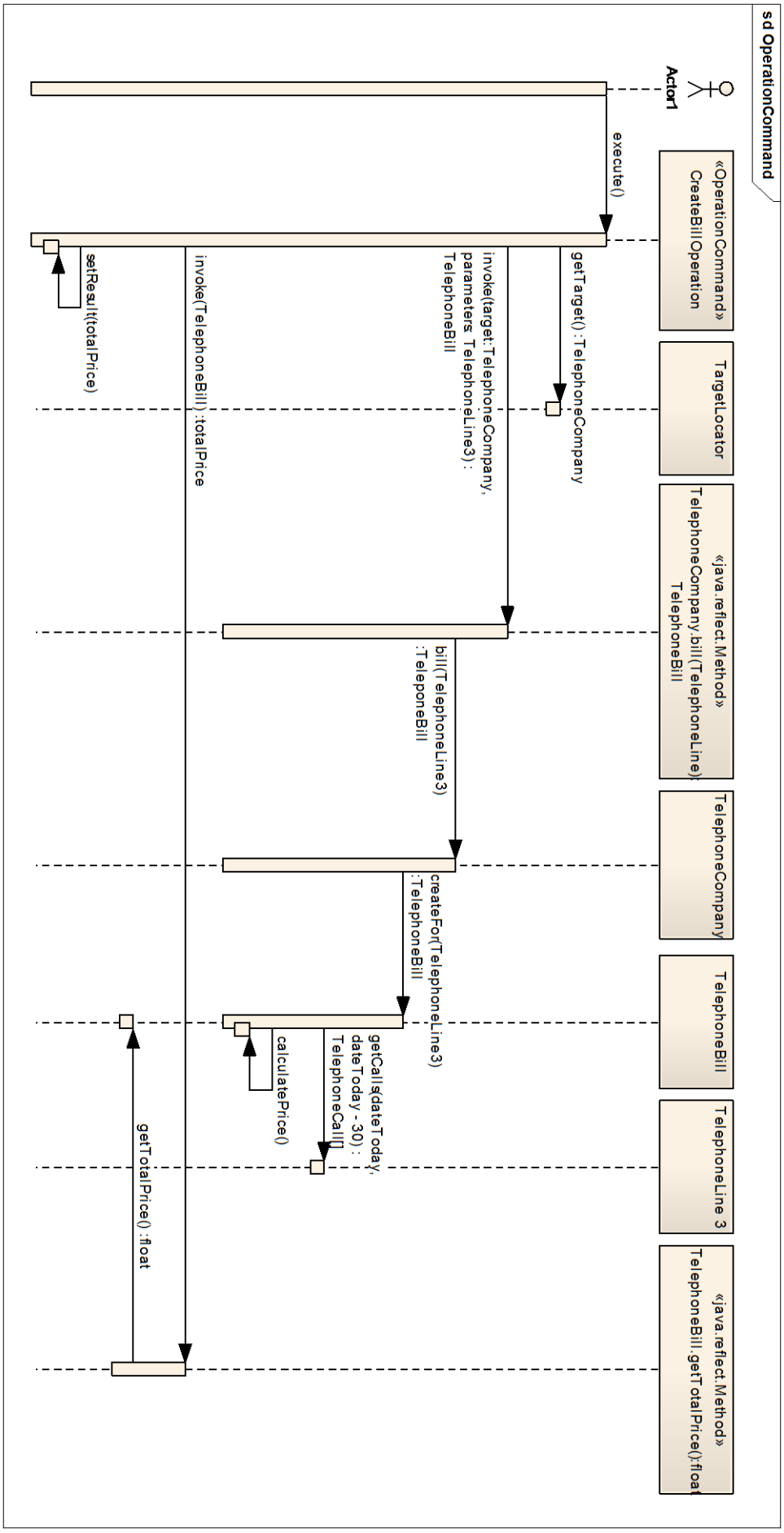


Figura 5.9 – Instanciación concreta con Target Locator y operaciones de acceso al resultado.

En el siguiente Diagrama de Interacción se describe la dinámica de los elementos anteriores en el contexto de la ejecución de la operación.


sd OperationCommand
Actor1
«OperationCommand»
CreateBillOperation
TargetLocator
«java.reflect.Method»
TelephoneCompany.bill(TelephoneLine)
TelephoneBill
TelephoneCompany
TelephoneBill
TelephoneLine 3
«java.reflect.Method»
TelephoneBill.getTotalPrice():float
execute()
getTarget() :TelephoneCompany
invoke(target:TelephoneCompany, parameters TelephoneLine3) : TelephoneBill
bill(TelephoneLine3) :TelephoneBill
createFor(TelephoneLine3) :TelephoneBill
getCalls(dateToday - 30) : TelephoneCall[]
calculatePrice()
invoke(TelephoneBill) :totalPrice
getTotalPrice() :float
setResult(totalPrice)

5.3.3. Operaciones de Ejecución y Aserción.

Tal como se ha descripto en secciones anteriores, antes de la ejecución de un test debe cargarse el juego de datos sobre el que la misma se realizará. DBUnit permite especificar este juego de datos mediante un archivo xml que contiene datos que pueden ser cargados directamente y de forma automática en la base de datos de la aplicación a testear. En nuestro modelo este juego de datos está representado mediante la clase DataSet, y es conocido por el DynamicTestCase de forma que pueda cargarlo ante cada ejecución.

Ya cargado el DataSet puede pasarse a la etapa de ejecución y aserción del test. Tal como se ha introducido en la sección anterior, las Operaciones de Ejecución son aquellas que se ejecutan sobre el juego de datos para testear determinada funcionalidad, mientras que las Operaciones de Aserción son las que se ejecutan inmediatamente a continuación, y permiten corroborar que los resultados obtenidos coinciden con los esperados. A continuación se describe cómo ambas son modeladas.

El concepto de Operación de Ejecución está cubierto y modelado mediante la clase OperationCommand previamente descripta, ya que esta última define una operación con sus parámetros y el objeto (target) sobre el cual se debe ejecutar. Por definición un test puede contener varias operaciones de ejecución, por lo que en nuestro modelo un DynamicTestCase conocerá una colección de OperationCommands, que ejecutará una tras otra en la secuencia definida.

El concepto de Operación de Aserción incluye la idea de operación representada por la clase OperationCommand pero, a diferencia de la Operación de Ejecución, define también otros aspectos. Éstos son: la especificación de un resultado esperado y la forma de comparación de ese resultado esperado con el resultado obtenido de la ejecución de la operación de aserción.

En la figura siguiente puede observarse la forma en que se permite la configuración de un DynamicTestCase con un juego de datos (DataSet) y ambos tipos de operaciones: ejecución y aserción.

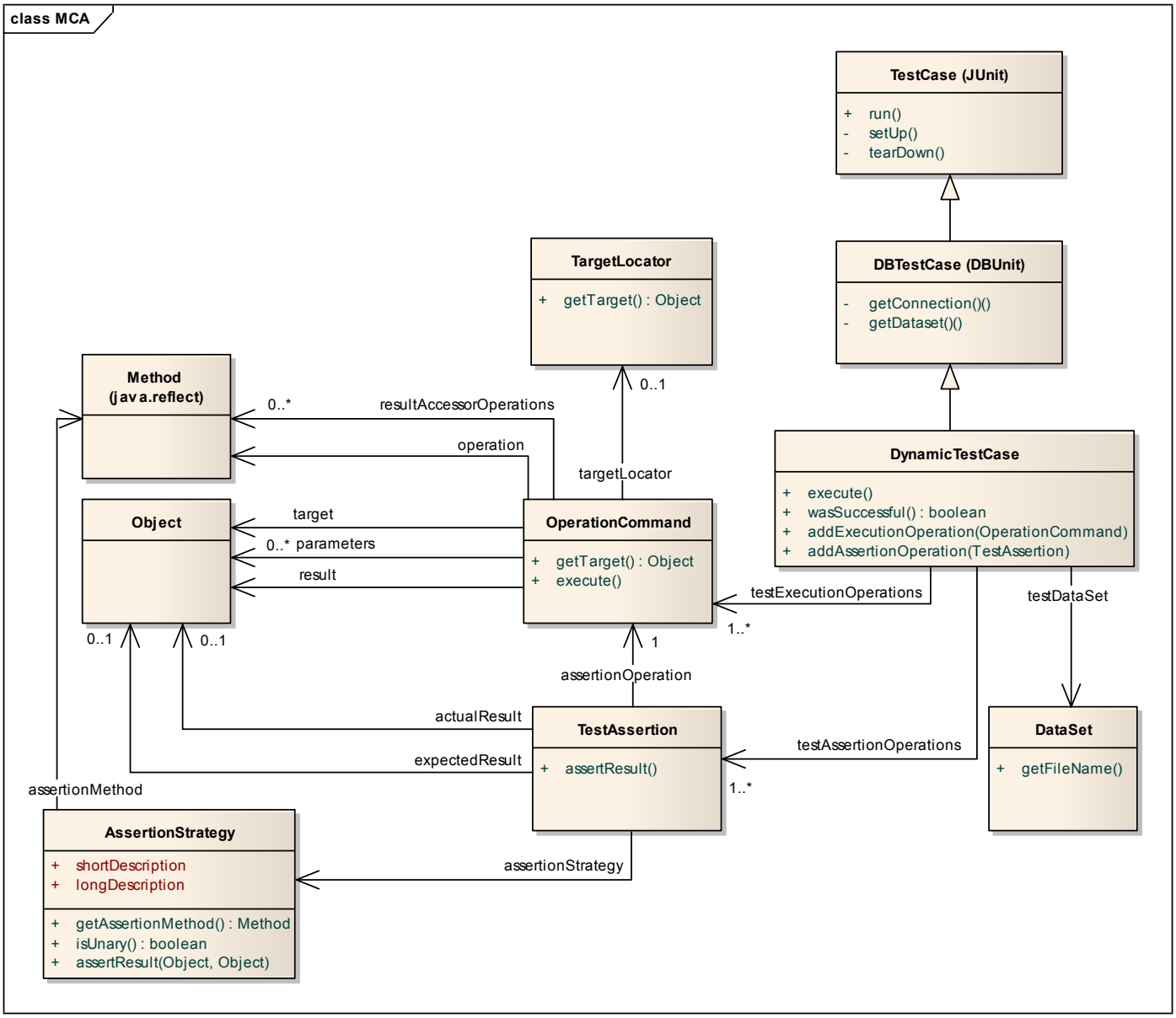


Figura 5.11 – Operaciones de Ejecución y Aserción

Aquí puede observarse que un test conoce un conjunto de Operaciones de Ejecución y un conjunto de Operaciones de Aserción, estando las primeras modeladas mediante la clase OperationCommand y las segundas mediante la clase TestAssertion.

Se observa que TestAssertion conoce a un OperationCommand, que representa la operación concreta que se ejecutará para realizar el assert a partir de su resultado y que, a su vez, como toda operación, conoce su target, mensaje a ejecutar y parámetros del mismo. Cabe destacar que aquí continúa operando implícitamente el mecanismo descripto previamente para los OperationCommands en cuanto a las operaciones de acceso al resultado (resultAccessorOperations), por lo que el resultado final de la operación es el obtenido luego de la ejecución en cascada de tales operaciones de acceso.

Este resultado obtenido se constituye en parámetro del método de aserción que se haya configurado. Estos métodos son los ya definidas por JUnit y DBUnit en la clase Assertion y pueden ser unarios o binarios. Entre los primeros, JUnit define los siguientes:

- Null. Verifica que el resultado sea nulo.

- Not Null. Verifica que el resultado no sea nulo.
- True. Verifica que el resultado es verdadero (lógica booleana).
- False. Verifica que el resultado es false (lógica booleana).

Como se ve, los métodos unarios toman un solo parámetro, que es el resultado obtenido de la ejecución de la operación de aserción. Por el contrario, los métodos binarios toman un segundo parámetro, el resultado esperado, de forma de poder comparar a ambos y así determinar el éxito o falla de la operación de aserción. Entre otros, JUnit define los siguientes métodos binarios:

- Equals. Verifica que el resultado obtenido sea igual al resultado esperado.
- Not Equals. Verifica que el resultado obtenido no sea igual al resultado esperado.

Dado que la elección del método de aserción es realizada por el usuario final en tiempo de ejecución debe encontrarse una forma genérica y homogénea de tratamiento de todos estos estos métodos, sean unarios o binarios. En nuestro modelo, esto se consigue mediante la clase AssertionStrategy. Haciendo uso de reflection, a esta clase se la configura con alguno de los métodos de aserción arriba descriptos y definidos por JUnit en su clase Assertion. A su vez, AssertionStrategy define un único mensaje general de aserción que siempre recibe dos parámetros, el resultado obtenido y el resultado esperado, pudiendo éste último ser nulo. De esta manera el TestAssertion, luego de ejecutar la operación, siempre delega la responsabilidad de realizar la aserción al AssertionStrategy, comunicándose con él siempre de la misma manera, lo que le permite un tratamiento homogéneo de todos ellos ahorrándole en absoluto la necesidad de conocer de qué tipo de estrategia concreta se trata. Esto permite mayor flexibilidad del modelo ya que el mismo puede ser extendido fácilmente con nuevas estrategias de aserción, con el único de requisito de que implementen un mensaje determinado (assertResult).

Para mantener la flexibilidad del modelo que es necesaria para su configuración en tiempo de ejecución, se define que el resultado esperado del TestAssertion es un Object.

Con todo esto cabe repasar la secuencia de ejecución de un DynamicTestCase, que se inicia cuando este test recibe el mensaje execute():

1. Carga del DataSet definido para el DynamicTestCase.
2. Ejecución de las Operaciones de Ejecución
3. Ejecución de las Operaciones de Aserción. Para cada una de ellas:
    a. Ejecución de la operación que define (OperationCommand), lo cual genera un resultado.
    b. Delegación de la ejecución de la aserción al AssertionStrategy, pasándole el resultado obtenido y, si la estrategia es binaria, también el resultado esperado concreto.

Cabe destacar que tanto las Operaciones de Ejecución y Aserción se continúan ejecutando mientras en ninguna de ellas se produzca un error, en cuyo caso se aborta la ejecución del test y se lo considera fallado. De la misma manera, la ejecución de las Operaciones de Aserción se corta ni bien una de ellas no pase la aserción de un resultado, dando así por fallado el test.

Toda esta dinámica se muestra en detalle en el siguiente diagrama.

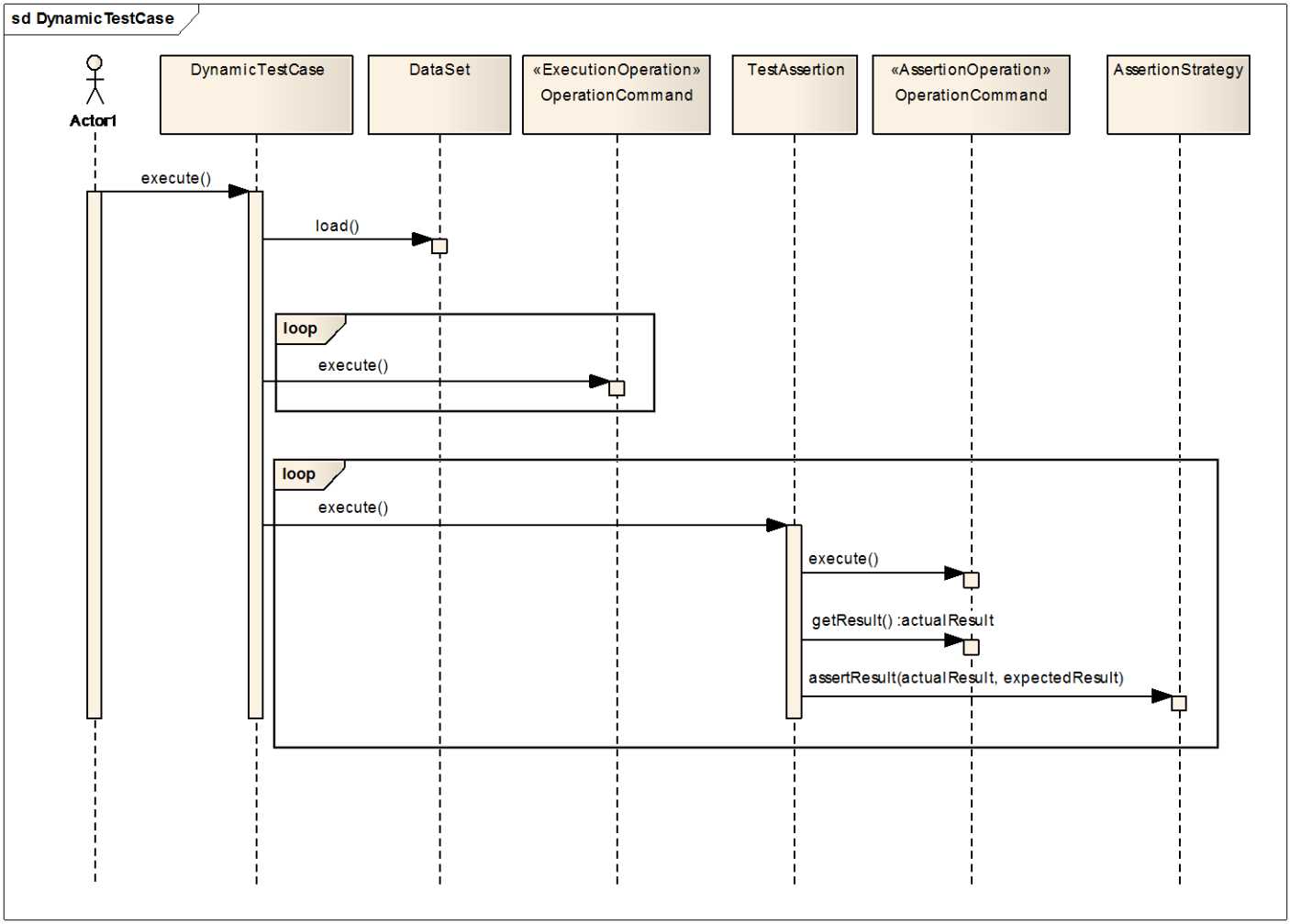


Figura 5.12 – Ejecución de un DynamicTestCase

5.3.4. Operaciones de Aserción Grupales

En el estado de evolución del modelo de la sección anterior, las Operaciones de Aserción definen una operación determinada, una estrategia de aserción particular y, eventualmente, un objeto concreto como resultado esperado. A este tipo de aserción podemos llamarla simple, o puntual, ya que el objetivo es realizar una aserción entre dos objetos puntuales, bien definidos: un objeto obtenido como resultado y otro objeto definido como resultado esperado.

Además de este tipo de aserciones, DBUnit define "aserciones grupales", las cuales no se realizan sobre objetos determinados sino sobre un área completa de la base de datos. De esta manera, luego de la ejecución de las operaciones de ejecución, puede realizarse la aserción de que los datos de toda un área determinada de la base de datos sean iguales a los esperados por el usuario, quien previamente los ha especificado. Esta especificación del estado esperado de un área se realiza, de forma similar al DataSet, mediante un archivo xml, pudiendo confeccionarse manualmente o mediante exportación.

En línea con lo anterior, y a diferencia de una aserción simple, una aserción grupal no define ni una operación ni una estrategia de aserción concreta, y el resultado esperado ya no es un objeto en particular sino un área completa de la base de datos (assertionDataset).

En la siguiente figura se muestra una nueva evolución del modelo, ahora contemplando la posibilidad de definir tanto operaciones de aserción simples como grupales.

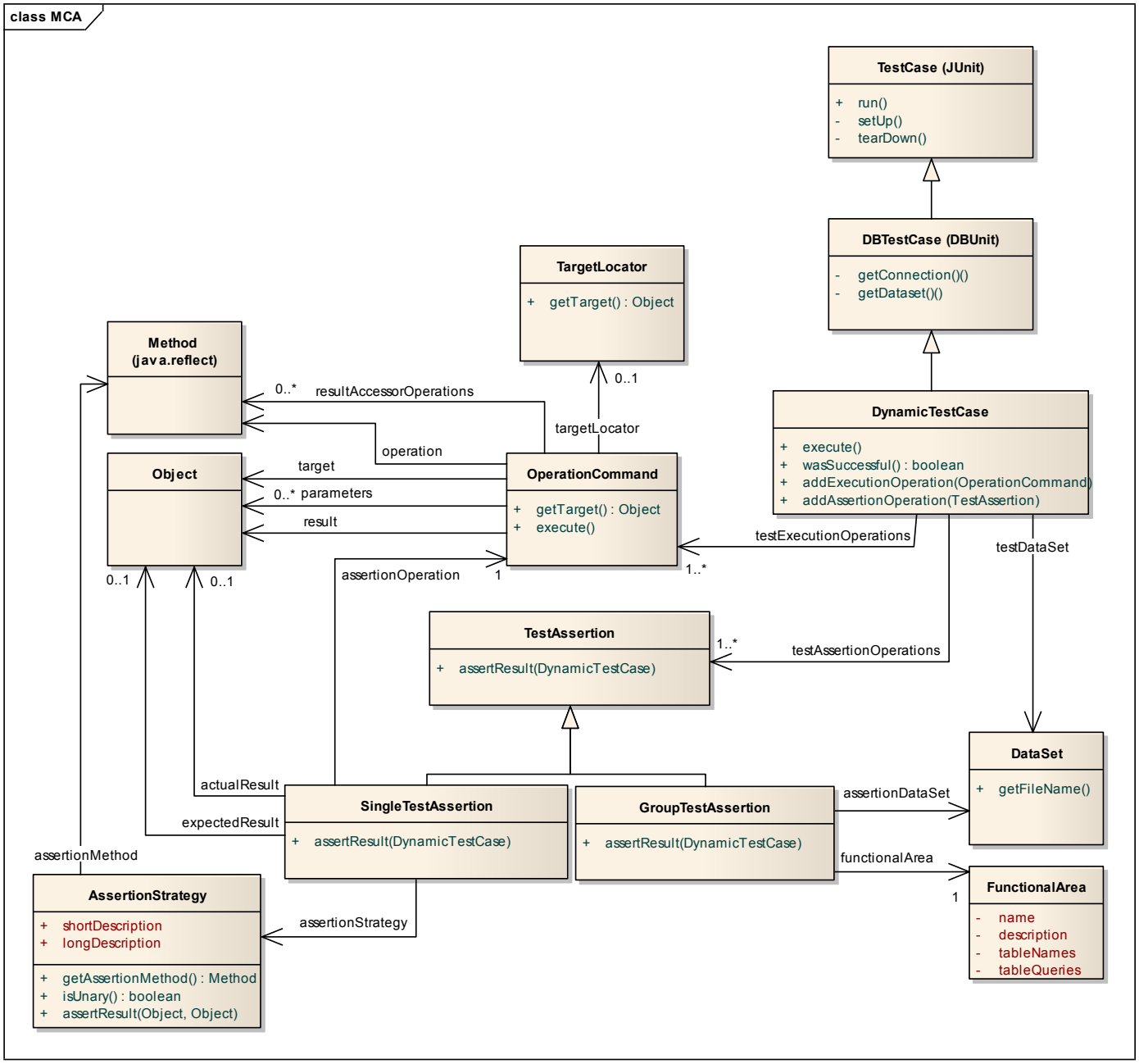


Figura 5.13 – Operaciones de Aserción Grupales

Cabe destacar el tratamiento homogéneo que DynamicTestCase realiza con los TestAssertions, ya que se comunica con un protocolo polimórfico tanto con los SingleTestAssertions como con los GroupTestAssertions.

Debe notarse también la diferencia de roles con que DynamicTestCase y GroupTestAssertions conocen a DataSet. Para el primero, el DataSet es el juego de datos que debe cargarse inmediatamente antes de la ejecución del test, mientras que para el segundo, el DataSet (otra instancia diferente) representa el juego de datos contra el que debe compararse un área de la base de datos luego de la ejecución del test (etapa de aserción).

El GroupTestAssertion conoce el área de la base de datos sobre la cual debe realizar la aserción: FunctionalArea. Las instancias de esta clase referencian un conjunto de tablas de la base de datos relacional que tienen una cierta cohesión funcional, es decir que conforman una cierta unidad en cuanto a los datos que contienen. Esto permite que puedan pre-definirse algunas pocas áreas determinadas que puedan ser reutilizadas las veces que sea necesario.

Existiendo la aserción grupal, que realiza la aserción de áreas completas de la base de datos, hay lugar para preguntar si tiene sentido que un test pueda conocer también, al mismo tiempo, a aserciones simples (SingleTestAssertions). Se tomó la decisión de diseño de que efectivamente esto tiene sentido, ya que en un test podría ser necesario realizar una aserción de una determinada área completa, más algunas aserciones puntuales en funcionalidad con base en otras áreas del sistema.

Con base a lo anterior, la secuencia de ejecución de un DynamicTestCase, que se inicia cuando este test recibe el mensaje execute() es la siguiente:

- Carga del DataSet con el juego de datos (testDataSet) definido para el DynamicTestCase.
- Ejecución de las Operaciones de Ejecución
- Ejecución de las Operaciones de Aserción. Para cada una de ellas:
  - Si es un SingleTestAssertion:
    - Ejecución de la operación que define (OperationCommand), lo cual genera un resultado.
    - Delegación de la ejecución de la aserción al AssertionStrategy, pasándole el resultado obtenido y, si la estrategia es binaria, también el resultado esperado concreto.
  - Si es un GroupTestAssertion:
    - Carga del DataSet que contiene los datos esperados (assertionDataSet) del área de base de datos sobre la cual se realizará la aserción.
    - Aserción por igualdad de todos los datos de DataSet cargado en el punto anterior que corresponde al área FunctionalArea especificada.

De la misma forma que lo descripto en la sección anterior, la ejecución del test se aborta y el mismo se considera fallado tan pronto como una operación de ejecución o aserción arroje un error o una de estas últimas falle en el chequeo.

5.3.5. Test Suites

En la sección anterior se concluyó la explicación completa de la ejecución de un test individual. Los tests individuales pueden agruparse por algún criterio determinado formando conjuntos de tests denominados Tests Suites. Un test suite, además de tests individuales puede contener otros test suites, conformando de esta manera una estructura de árbol. De esta forma, desde lo funcional, se puede ejecutar cualquier test individual del árbol como así también cualquier test suite. En este último caso, la ejecución del test suite implica la ejecución de los tests individuales que contiene como así también la ejecución de los test suite hijos, lo cual genera una ejecución recursiva hasta llegar a los nodos hoja.

La ejecución de un test suite se considera exitosa si la ejecución todos sus hijos (tests individuales y test suites) ha sido exitosa, y se considera fallado si la ejecución de al menos uno de ellos ha fallado.

En el modelo básico esta estructura arbórea se consigue mediante la instanciación del patrón de diseño Composite, el cual permite no sólo lograr esta estructura sino que también permite al cliente de la misma un tratamiento homogéneo de sus componentes, sin hacer diferencias en cuanto a la naturaleza estructural de cada una (simple, como el test individual, o compuesta, como un test suite). El patrón Composite especifica que en este caso se creen dos clases, una para el test individual y otra para el test compuesto (o sea el test suite), más una

tercera que sea superclase de las dos anteriores y que represente la abstracción de ambas definiendo la estructura y protocolo comunes a las dos. Esta tercera clase representaría simplemente el concepto de test, ya que un test suite también puede considerarse un test, sólo que algo más complejo debido a su composición.

En línea con la estructura descripta y especificada por el patrón Composite, en el modelo básico el concepto test simple estaría cubierto por la clase DynamicTestCase, mientras que debe crearse una nueva clase TestSuite para el concepto de test compuesto, y una superclase TestUnit que abstraiga estructura y comportamiento comunes. En la siguiente figura se muestra cómo se aplicaría el patrón Composite a esta parte del modelo.

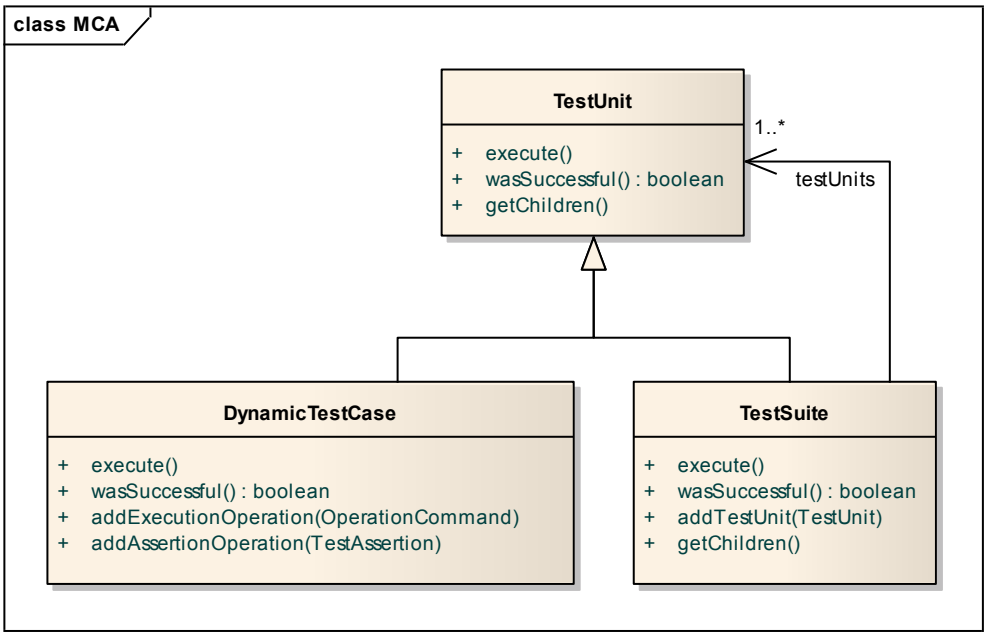
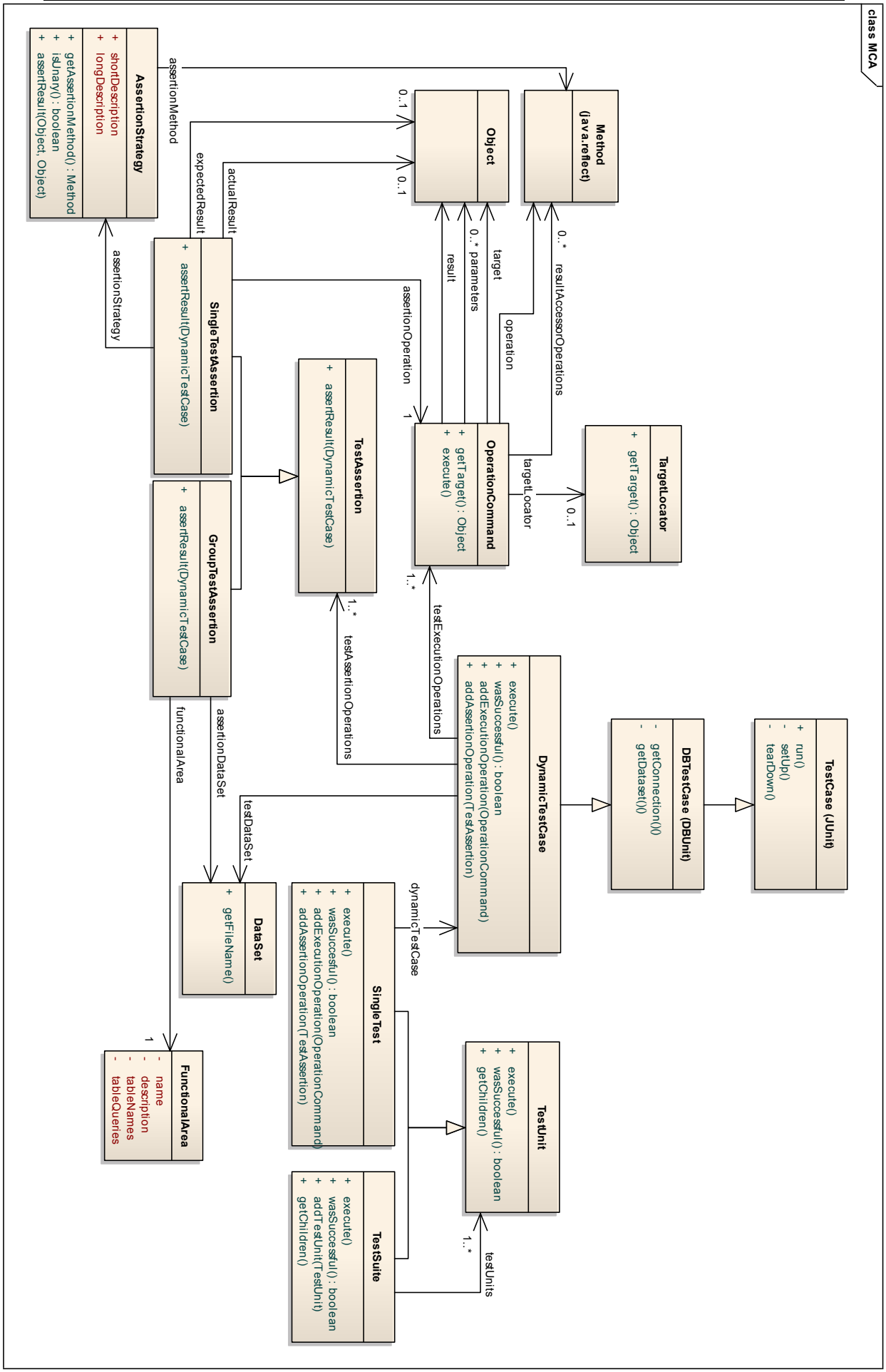


Figura 5.14 – Test Suites mediante el Patrón Composite

Si bien la aplicación directa del patrón Composite sería la de la figura anterior, esto no es posible en nuestro modelo si es que consideramos que no trabajamos en un contexto que permita herencia múltiple de clases (tal es el caso de muchos lenguajes de programación orientada a objetos, como Java y Smalltalk). Sucede que, tal como se describió en secciones anteriores, DynamicTestCase ya tiene una superclase (DBTestCase) de quien hereda esctructura y comportamiento necesarios para su esencia de test y, al trabajar con herencia simple, no se permitiría que tuviera a TestUnit como segunda superclase.

Como solución al problema anterior se tomó la decisión de crear una nueva clase, SingleTest, que conozca a la clase DynamicTestCase y que delegue a ésta todas las operaciones correspondientes a tests individuales. De esta forma, DynamicTestCase seguirá conservando su superclase original mientras que SingleTest tendrá como superclase a TestUnit. Esta solución es parecida a la propuesta por el patrón Adapter ya que SingleTest no implementa comportamiento y sólo delega a otra clase las operaciones que le solicitan. Sin embargo, esta clase no realiza una transformación de interfaces (objetivo del patrón Adapter) sino que re-transmite los mensajes tal cual le llegan. Esta solución tiene una estructura y dinámica parecida a la definida por tal patrón, pero se entiende que no es una aplicación del mismo.

En la figura siguiente se muestra la evolución del modelo incluyendo la gestión de test suites descripta.


class MCA
TestCase (JUnit)
+ run()
- setUp()
- tearDown()
DBTestCase (DBUnit)
- getConnection()()
- getDataset()()
DynamicTestCase
+ execute()
+ wasSuccessful() : boolean
+ addExecutionOperation(OperationCommand)
+ addAssertionOperation(TestAssertion)
TestUnit
+ execute()
+ wasSuccessful() : boolean
+ getChildren()
SingleTest
+ execute()
+ wasSuccessful() : boolean
+ addExecutionOperation(OperationCommand)
+ addAssertionOperation(TestAssertion)
TestSuite
+ execute()
+ wasSuccessful() : boolean
+ addTestUnit(TestUnit)
+ getChildren()
DataSet
+ getFileName()
FunctionalArea
- name
- description
- tableNames
- tableQueries
TargetLocator
+ getTarget() : Object
OperationCommand
+ getTarget() : Object
+ execute()
TestAssertion
+ assertResult(DynamicTestCase)
SingleTestAssertion
+ assertResult(DynamicTestCase)
GroupTestAssertion
+ assertResult(DynamicTestCase)
AssertionStrategy
+ shortDescription
+ longDescription
+ getAssertionMethod() : Method
+ isUnary() : boolean
+ assertResult(Object, Object)
Method (java.reflect)
Object
assertionMethod
expectedResult
actualResult
assertionStrategy
assertionOperation
result
parameters
target
operation
resultAccessorOperations
targetLocator
testExecutionOperations
testAssertionOperations
testDataSet
assertionDataSet
functionalArea
dynamicTestCase
testUnits
0..1
0..*
1..*
1

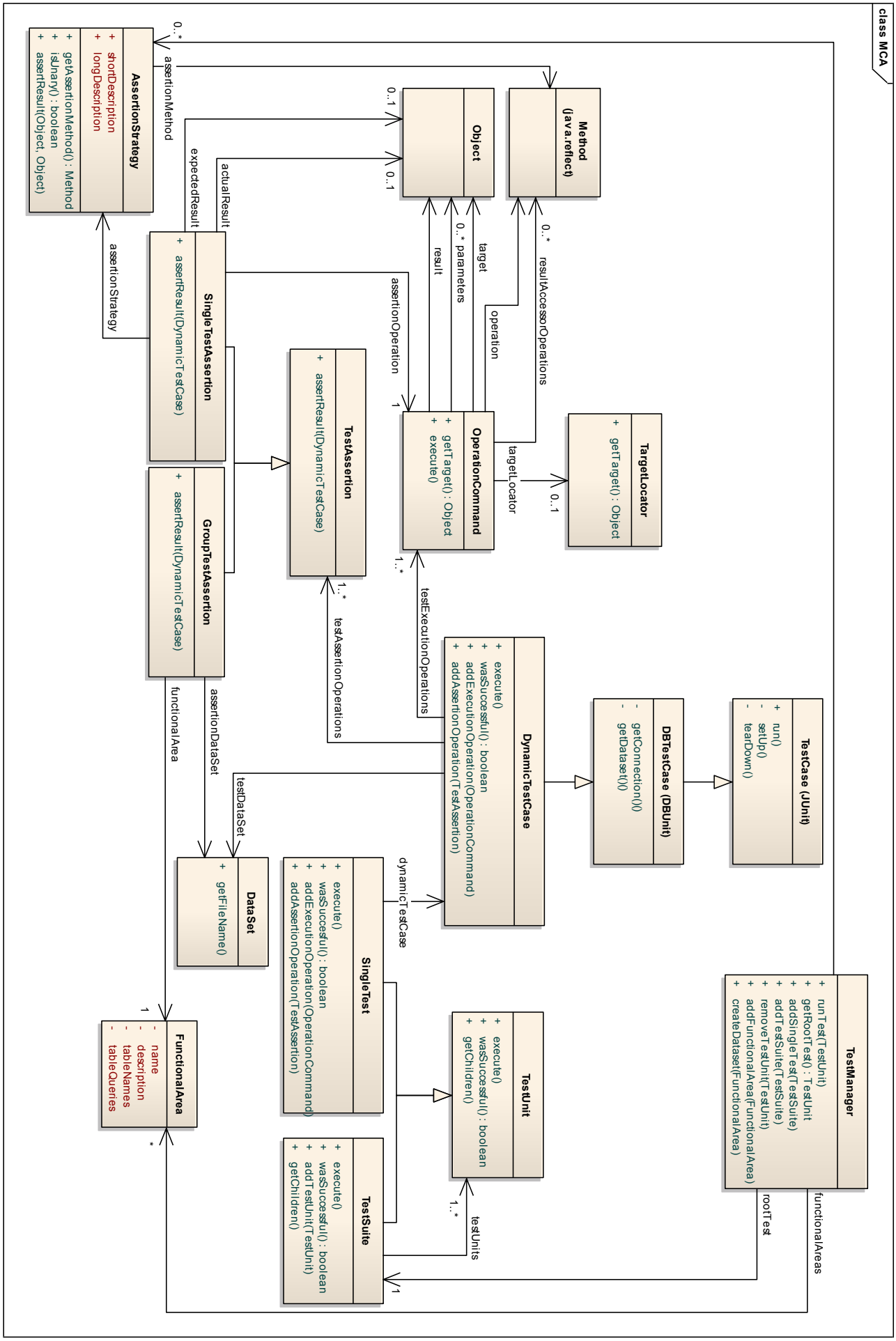
5.3.6. Modelo básico completo

El modelo descripto en la sección anterior permite una gestión integral de tests según el objetivo de este trabajo. No obstante esto, debe existir una clase que se encargue de la gestión integral del sistema en el sentido de que pueda relacionar y hacer interactuar entre sí a las instancias de las clases ya descriptas. Esto se consigue aplicando el patrón de diseño Facade, el cual provee al usuario del modelo un único punto de entrada al mismo, evitándole realizar demasiadas operaciones detalladas con las distintas componentes del mismo.

En este sentido se crea la clase TestManager que, entre otras, tiene las siguientes responsabilidades:

- Mantener una referencia al nodo raíz (test suite) del árbol de tests.
- Mantener todas las áreas funcionales disponibles en el modelo.
- Mantener todas las estrategias de aserción disponibles en el sistema.
- Exportación de datasets, tanto de para uso como juego de datos como áreas de la base de datos en aserciones grupales.
- Protocolo básico que permita al usuario la gestión de tests: creación, borrado, estructuración en árbol, ejecución y consulta de estado, entre otras.

En la figura siguiente se muestra el modelo básico completo.


class MCA
TestManager
+ runTest(TestUnit)
+ getRootTest() : TestUnit
+ addSingleTest(TestSuite)
+ addTestSuite(TestSuite)
+ removeTestUnit(TestUnit)
+ addFunctionalArea(FunctionalArea)
+ createDataset(FunctionalArea)
functionalAreas
rootTest
TestCase (JUnit)
+ run()
- setUp()
- tearDown()
DBTestCase (DBUnit)
- getConnection()()
- getDataset()()
DynamicTestCase
+ execute()
+ wasSuccessful() : boolean
+ addExecutionOperation(OperationCommand)
+ addAssertionOperation(TestAssertion)
testExecutionOperations
testAssertionOperations
dynamicTestCase
TestUnit
+ execute()
+ wasSuccessful() : boolean
+ getChildren()
testUnits
SingleTest
+ execute()
+ wasSuccessful() : boolean
+ addExecutionOperation(OperationCommand)
+ addAssertionOperation(TestAssertion)
TestSuite
+ execute()
+ wasSuccessful() : boolean
+ addTestUnit(TestUnit)
+ getChildren()
DataSet
+ getFileName()
testDataSet
assertionDataSet
FunctionalArea
- name
- description
- tableNames
- tableQueries
functionalArea
TestAssertion
+ assertResult(DynamicTestCase)
SingleTestAssertion
+ assertResult(DynamicTestCase)
GroupTestAssertion
+ assertResult(DynamicTestCase)
assertionOperation
assertionStrategy
AssertionStrategy
+ shortDescription
+ longDescription
+ getAssertionMethod() : Method
+ isUnary() : boolean
+ assertResult(Object, Object)
assertionMethod
expectedResult
actualResult
OperationCommand
+ getTarget() : Object
+ execute()
target
operation
parameters
result
resultAccessorOperations
targetLocator
TargetLocator
+ getTarget() : Object
Object
Method
(java.reflect)
0..*
0..1
1
1..*

6. MODELO EXTENDIDO DE LA APLICACIÓN DE TESTING PARA SOA

El modelo básico descripto en el capítulo anterior es de uso genérico y puede aplicarse en una amplia variedad de contextos, tal como se ha descripto en los lineamientos de diseño.

En este capítulo y en el siguiente se describe la adaptación del modelo básico a ciertas tecnologías particulares, que son las utilizadas en una gran cantidad de sistemas de la industria en la actualidad, entre ellos el de la TGP.

A continuación se describen las técnicas de extensión que se utilizarán para este objetivo y los fundamentos de su elección. Posteriormente se describe la extensión y configuración del modelo básico para su uso con Aplicaciones Objetivo de tipo SOA, mientras que en el capítulo siguiente se describirán la extensión que permite que el modelo básico de la Aplicación de Testing pueda ser persistido de forma transparente.

6.1. Técnicas de Extensión

En esta sección se describen los lineamientos de diseño que se toman como eje para el diseño de las extensiones como así también las formas de implementación que permiten llevarlos a cabo.

6.1.1. Lineamientos de Diseño para la Extensión

Para la extensión del modelo básico también se sigue un lineamiento de diseño fundamental:

- La funcionalidad de extensión debe consistir sólo en un agregado transparente al modelo básico y no en una modificación del mismo.

El objetivo de esto que la extensión sea sencilla y requiera sólo una configuración (tipo plug-in) y no esfuerzo de desarrollo de programadores. Por otro lado, podrían existir otras muchas extensiones para diversos contextos particulares, y todas ellas deberían poder montarse sobre el mismo y único modelo básico de forma transparente.

6.1.2. Implementación de la extensión

De forma tal de cumplir con el lineamiento de diseño mencionado en la sección anterior, se utilizarán las dos técnicas de extensión que se describen a continuación. No obstante esto, más adelante se describirán los contextos y formas concretas en donde éstas se utilizan.

Interceptors:

Se implementan mediante subclasificación (técnica White-box) de una clase del modelo básico con una subclase que pertenece a la extensión. La subclase define un mensaje con el mismo nombre que su superclase, y en el método que implementa invoca al método de su superclase, pero realizando acciones adicionales antes o después de hacerlo.

Esta estrategia no es intrusiva (tal como lo requiere el lineamiento de diseño definido) ya que, mediante el principio de sustitución de Liskov, una instancia de la subclase puede reemplazar a una instancia de su superclase, por lo cual el modelo básico continuará funcionando normalmente.

Composición:

La extensión por composición es una técnica black-box mediante la cual se le provee al modelo a extender un objeto que cumple cierto protocolo y resuelve las operaciones que aquél necesita. La flexibilidad de la extensión viene dada porque para cada contexto particular, puede crearse un objeto diferente que, para las mismas operaciones, provea implementaciones diferentes y acordes al contexto particular. En términos de patrones de diseño, en este trabajo esto se implementa mediante strategies, factories y commands.

Ambas técnicas son muy utilizadas en la instanciación de frameworks, en donde se debe extender un "core" con funcionalidad más concreta y de forma transparente, sin modificarlo [RalphJ]. Este objetivo es el mismo que se persigue en la extensión que se pretende realizar aquí.

6.2. Configuración del Modelo Básico para aplicación objetivo SOA

En esta sección se describe la forma en que el modelo básico previamente descripto puede ser utilizado en un contexto de tipo SOA. En este sentido se describen las extensiones y configuraciones requeridas para lograr este objetivo.

6.2.1. Configuración de Operaciones de Servicios

El Modelo Básico antes descripto puede configurarse de forma directa y sin que se requiera extensión alguna para el testeo de aplicaciones objetivo de tipo SOA como las ya descriptas. En esta sección se describe la forma en que, sin ser alterado, el modelo básico puede configurarse para utilizarse en tal contexto.

Tal como se ha explicado previamente, los servicios de la aplicación objetivo están representados mediante objetos que los implementan. De esta forma, la única manera de interactuar con el modelo es través de las operaciones que estos objetos proveen (exportan). A un nivel de detalle más bajo, esto quiere decir que para ejecutar una operación de la aplicación objetivo se debe enviar un mensaje (operación) a un objeto (servicio). Y esto último es justamente un caso particular de la funcionalidad que el modelo básico provee y automatiza.

La configuración de una Operación de Ejecución sobre una aplicación objetivo de tipo SOA se realiza simplemente configurando un OperationCommand de la siguiente manera:

- Target = objeto servicio provisto por la aplicación objetivo.
- Operation = método del objeto servicio (target) que implementa la operación que se desea ejecutar.
- Parameters = parámetros de la operación anterior
- ResultAccessorOperations = eventual lista de operaciones (mensajes, en particular y generalmente getters) que permiten acceder al objeto devuelto por la ejecución de la operación.
- Result = objeto obtenido como resultado final de la operación, es decir: es el resultado de la operación o, de haber especificadas resultAccessorOperations, el resultado final de aplicar tal secuencia de operaciones al resultado de la operación principal (operation). Cualquiera sea el caso, este resultado final se setea de forma automática e inmediata luego de la ejecución.

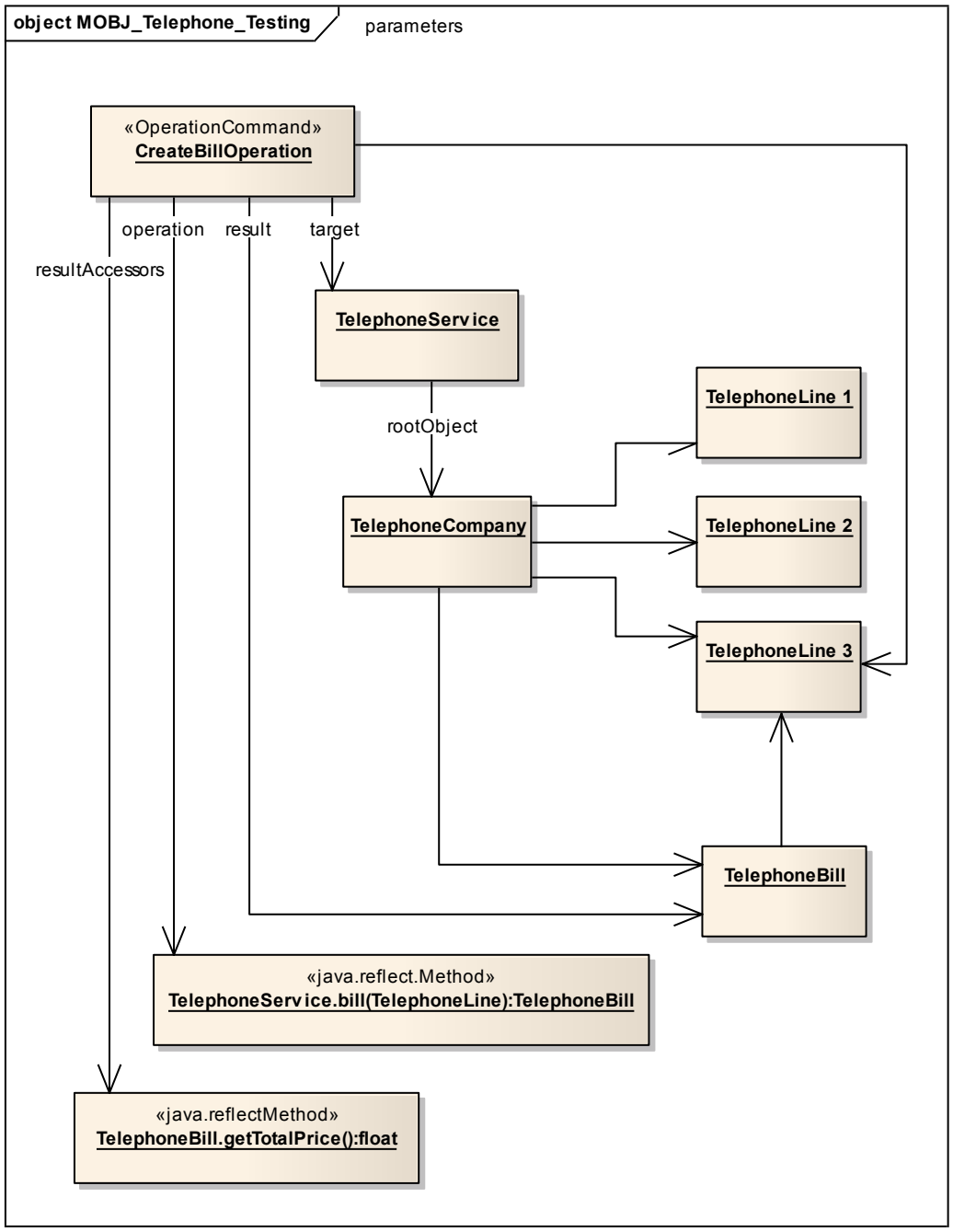


Figura 6.1 – Ejecución de operaciones en el contexto SOA

Esta configuración del modelo básico permite el trabajo servicios como se espera, sin embargo surge un inconveniente: ¿cómo se obtiene el objeto servicio de la aplicación objetivo?

6.2.2. Localización de Servicios de la aplicación objetivo

Una alternativa es que este objeto servicio (que se constituye en target del OperationCommand) pueda ser configurado al inicio del testing, esto es: que alguna aplicación de "arranque" pida el objeto servicio a la aplicación objetivo y luego lo setee como target al OperationCommand.

Esta una alternativa válida pero que sin embargo plantea el siguiente inconveniente: hay casos en que el objeto servicio provisto por la aplicación objetivo "caduca", es decir representa

una versión inválida en relación a tal aplicación. Esto sucede, por ejemplo, cuando la operación objetivo se reinicia o cuando el objeto servicio cambia su versión debido a alguna modificación, por ejemplo extensiva. Como se verá más adelante, este será justamente el contexto normal de uso cuando los tests puedan persistirse y ser reutilizados continuamente a lo largo de extensos períodos de tiempo.

En estos casos, es necesario obtener una versión actualizada del objeto servicio y volver a reconfigurar todos los OperationCommand con la nueva versión. Como puede apreciarse, el costo de mantenimiento es alto.

Una alternativa más adecuada sería que el OperationCommand siempre pueda obtener una versión actualizada del objeto servicio por sus propios medios, sin demandar que aplicaciones externas se tomen el trabajo de hacer esto por él. Como se ha explicado en secciones anteriores, esta funcionalidad de autonomía en cuanto a la localización del target es provista por el modelo básico mediante el TargetLocator. El OperationCommand simplemente pedirá el target a éste, quien siempre le devolverá la versión actualizada del objeto servicio que necesita.

Hasta aquí se ha quitado la responsabilidad de localización del servicio primero de aplicaciones externas, y también del OperationCommand, delegándola en el TargetLocator. Así se consigue una mejor distribución de responsabilidades y un modelo más autónomo. A continuación se describe cómo este TargetLocator, finalmente, lleva adelante esta localización del objeto servicio.

Tal como se ha explicado en la sección correspondiente al Modelo Básico, para obtener el target el OperationCommand simplemente envía el mensaje getTarget() al TargetLocator que tenga configurado. Esto permite configurar a un OperationCommand con una amplia variedad de TargetLocators, ya que aquél se comunica éste mediante tal mensaje polimórfico. Existe una posibilidad de implementación del TargetLocator que se entiende que cubre un amplio rango de necesidades. Esta alternativa es configurar a un objeto TargetLocator con dos atributos: locatorClassName y locatorMethodName.

El atributo locatorClassName es el nombre de la clase de la aplicación objetivo que implementa las acciones necesarias para conseguir el objeto target. Por decisión de diseño se asume que esta clase implementa el patrón de diseño Singleton, esto significa que existe una única instancia de esta clase y que la misma se obtiene enviándole a la clase el mensaje polimórfico “getInstance()”. Podría diseñarse una solución más abarcativa en donde tal clase no sea un Singleton y/o la instancia correspondiente se obtenga enviando un mensaje diferente; para esto simplemente bastaría agregar al TargetLocator un nuevo atributo con el nombre del mensaje que devuelva tal instancia.

En cuanto al atributo locatorMethodName, es el nombre del mensaje que se debe enviar a esa instancia obtenida (por lo tanto se trata de un mensaje de instancia) para que efectivamente devuelva el objeto target que el OperationComman necesita.

Aquí también se sigue el lineamiento de flexibilidad y dinamismo necesario e inherente a la naturaleza de este tipo de aplicaciones. En este sentido, estos nombres de clase y mensajes, que siempre corresponden a elementos de la aplicación objetivo, se especifican mediante inyección de dependencias vía un archivo externo de configuración (generalmente xml). De esta manera, el código de la aplicación de testing en este punto (TargetLocator) queda absolutamente desacoplado de la aplicación objetivo, no habiendo ningún tipo de indicio o rastro de la misma. Esto permite que la aplicación de testing se adapte fácil y dinámicamente a cualquier aplicación objetivo.

Habiendo conseguido esta generalización en cuanto a datos (nombres de clase y método), resta lograr la generalización del comportamiento, lo cual, nuevamente, se consigue mediante reflection. A continuación se muestra la secuencia de pasos que ejecuta el TargetLocator ante la recepción del mensaje getTarget() realizada por el OperationCommand:

1. A partir del nombre especificado en locatorClassName se obtiene la clase correspondiente.
2. A tal clase (que es un Singleton) se le pide su única instancia mediante el mensaje

"getInstance()".

3. Mediante reflection se envía a esa instancia el mensaje definido por el atributo locatorMethodName.
4. Se devuelve el objeto obtenido en el paso anterior, que es el target que el OperationCommand necesita.

Este diseño genérico del modelo permite configurarlo de manera directa para aplicaciones objetivo de tipo SOA como las que se describen en este trabajo. La configuración del TargetLocator para este tipo de aplicaciones se realiza de la siguiente manera:

- locatorClassName = nombre de la clase que actúa como ServiceLocator en la aplicación objetivo.
- locatorMethodName = nombre del método del ServiceLocator que devuelve el objeto servicio con el cual el OperationCommand necesita comunicarse.

De esta manera, mediante el envío del mensaje getTarget() al TargetLocator, el OperationCommand siempre obtiene una instancia actualizada del objeto servicio (su target) con quien debe comunicarse. Es a esa instancia (objeto servicio) a quien inmediatamente a continuación enviará el mensaje (operation) que efectivamente disparará la ejecución de la operación del servicio.

A continuación se brinda un ejemplo a los fines de mostrar el funcionamiento del modelo en el contexto concreto de la ejecución de un servicio sobre una aplicación de tipo SOA.

El ejemplo es una continuación del modelo de la Compañía Telefónica ya introducido, ahora extendido con servicios. En concreto, TelephoneService es quien exporta los servicios a los clientes que deseen utilizarlos, siendo el único mediador entre ellos y el modelo. Por su parte TelephoneServiceLocator es un objeto provisto por la aplicación de telefonía que permite obtener el objeto TelephoneService mencionado.

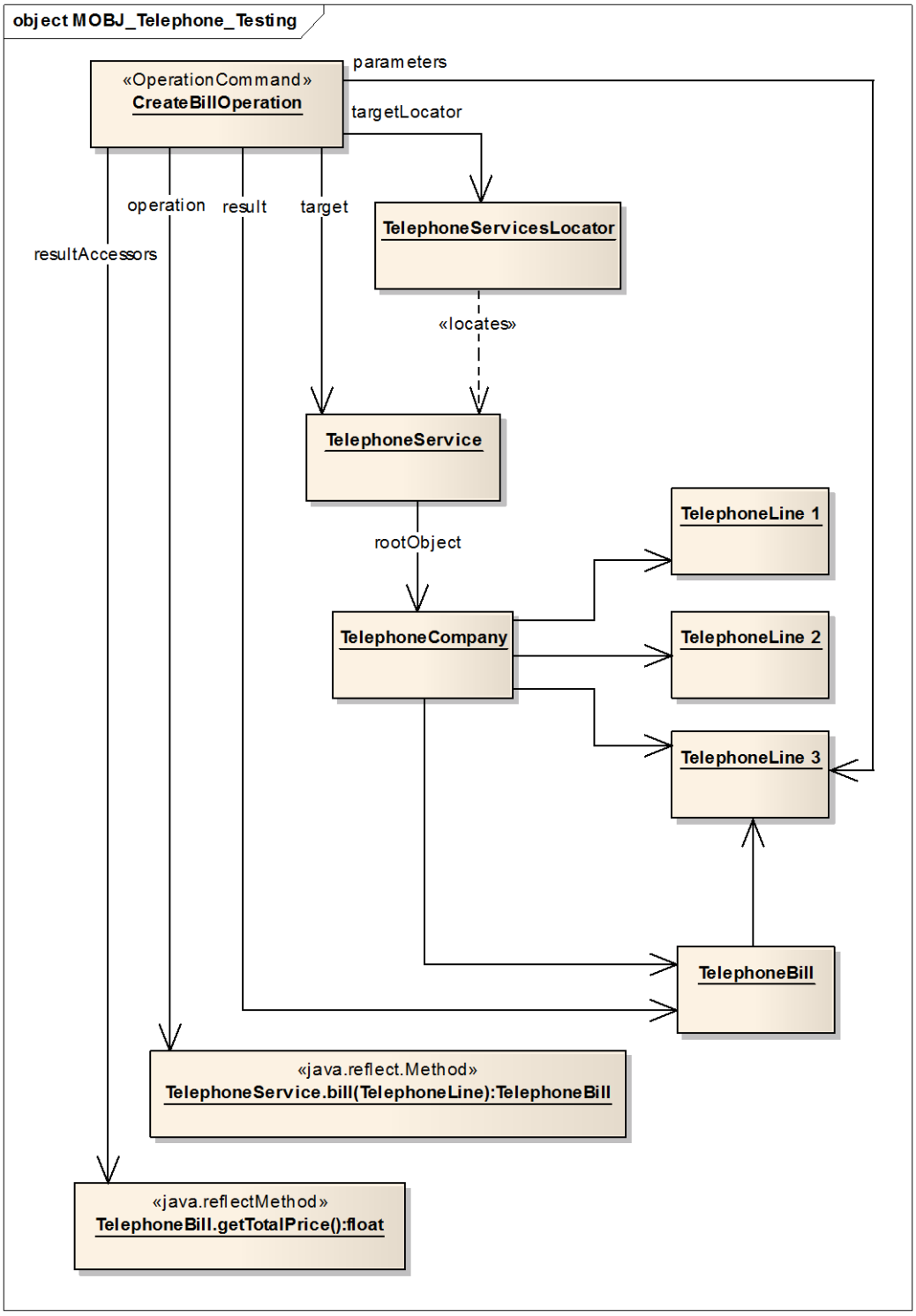


Figura 6.2 – Localización de servicios de la Aplicación Objetivo

6.2.3. Modelo de Descriptores de Servicios

Con base en lo explicado hasta aquí, el modelo básico puede configurarse para ser utilizado con servicios. Sin embargo, aún no se ha aclarado cómo un usuario final de la aplicación de testing podría llevar a cabo el tipo de configuraciones mencionados en la sección anterior. En este sentido, se hace necesario que tal persona, desde la aplicación de testing, pueda listar, estudiar y seleccionar los servicios, operaciones y parámetros que la aplicación

objetivo provee.

Para lograr esto último, se diseña el concepto de Descriptor. Así, en la aplicación de testing, se cargan descriptores de servicios, operaciones, parámetros y resultados que representan a los elementos de la aplicación objetivo, de forma que el usuario de la aplicación de testing pueda referenciarlos y combinarlos desde esta última.

A continuación se muestra el modelo de descriptores que se encuentra cargado en la aplicación de testing, pero que referencian elementos de la aplicación objetivo. Es muy importante aclarar que el usuario final verá solamente las descripciones de estos descriptores, y de ninguna manera verá ni tendrá acceso al resto de los atributos de índole técnica que se describen a continuación. No obstante lo anterior, estos atributos, aunque invisibles al usuario final, deben estar disponibles para la aplicación de testing ya que brindan los detalles técnicos necesarios para que esta aplicación funcione correctamente. Son necesarios, entre otras cosas, para el chequeo de tipos de parámetros, resultados y armado dinámico y automático de interfaces gráficas de acuerdo a la composición de cada uno de estos elementos. Todo esto será desarrollado en la sección dedicada a la interfaz gráfica de la aplicación de testing.

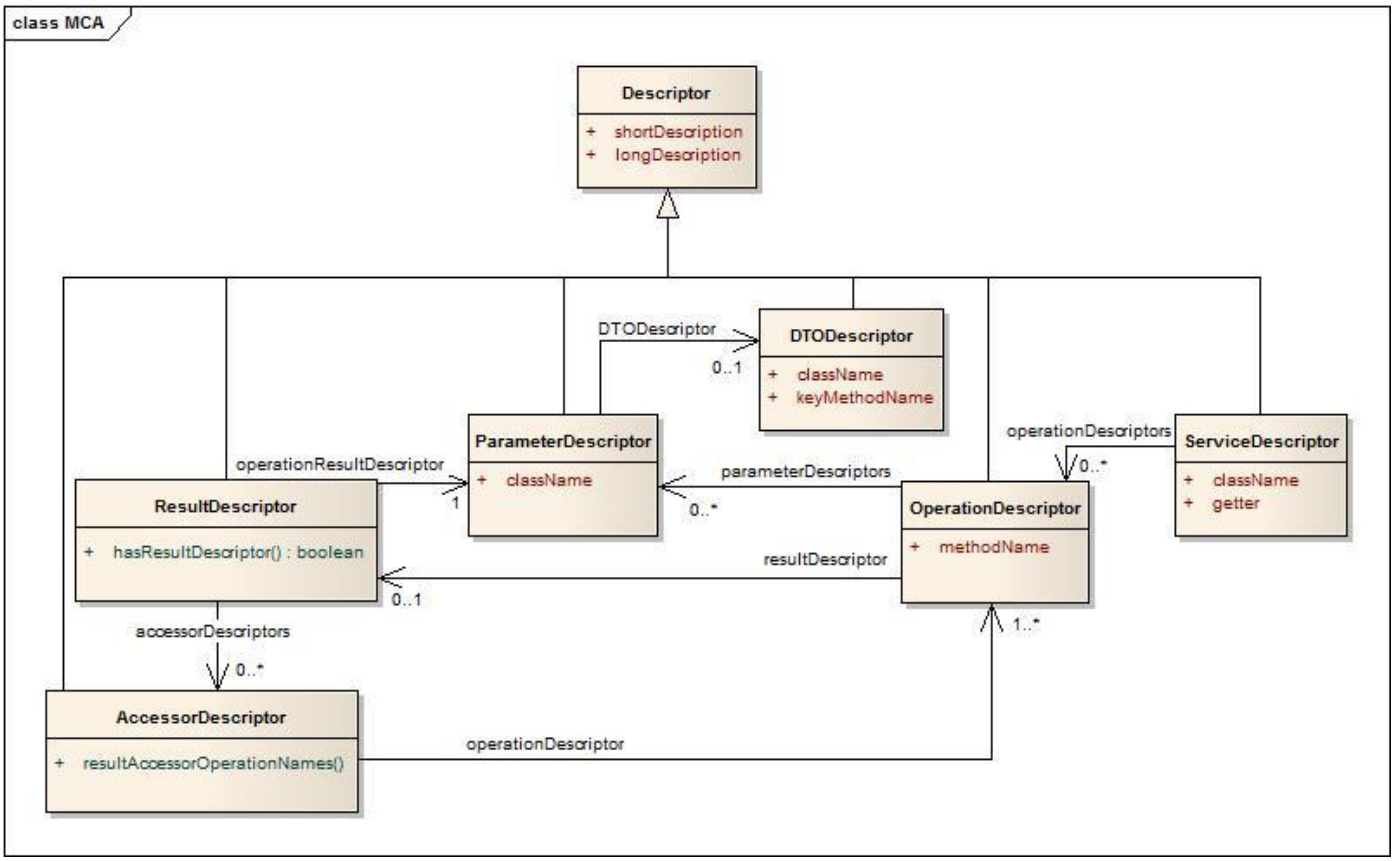


Figura6.3 – Modelo de Descriptores de Servicios

En la figura anterior puede observarse una superclase común a todos los Descriptores, Descriptor, que define atributos básicos a todos ellos como descripciones que serán útiles al usuario final al momento de la configuración.

Los descriptores se encadenan y relacionan entre sí, siendo el punto de partida la clase ServiceDescriptor. Esta clase define tres atributos:

- getter: es el nombre del mensaje getter con el cual se le pide al ServiceLocator de la

aplicación objetivo el objeto servicio que representa este descriptor.

- className: es la clase del objeto servicio que se obtiene como resultado de la ejecución del getter anterior; es decir, es la clase a la que pertenece el objeto servicio. Una instancia de esta clase será el objeto servicio concreto a quien el OperationCommand le enviará el mensaje de la operación final a ejecutar.
- operationDescriptors: es el conjunto de operaciones que se pueden ejecutar sobre este servicio (objeto servicio). Cada una de estas operaciones está especificada mediante un descriptor acorde a su naturaleza: OperationDescriptor.

En el ejemplo anterior, el getter sería el mensaje getTelephoneService(), implementado por la clase TelephoneServiceLocator); className sería TelephoneService; en cuanto a los operationDescriptors existiría al menos uno que describiría al método bill(TelephoneLine).

El OperationDescriptor especifica las operaciones que pueden ejecutarse para un servicio determinado. Para esto especifica los siguientes atributos:

- methodName: es el nombre del mensaje del objeto servicio que implementa la operación definida por este descriptor (OperationDescriptor).
- parameterDescriptors: es la lista de descriptores de parámetros de la operación implementada por el método anterior. Cada parámetro se especifica mediante un ParameterDescriptor, como se explica más adelante.
- resultDescriptor: es la descripción del resultado de la operación, y no se especifica si ésta no devuelve ningún resultado. Esta especificación se realiza mediante un ResultDescriptor, como se explica más adelante.

El ParameterDescriptor especifica un parámetro de una operación, incluyendo también la caracterización de su resultado. Para la descripción de un parámetro se especifica solamente su clase. Ésta constituye la solución más simple, ya que alternativas más evolucionadas podrían especificar otros aspectos, como por ejemplo rangos que tengan sentido para la operación en cuestión.

En caso de que un parámetro sea un DTO, el ParameterDescriptor podrá especificar el descriptor de este último mediante un DTODescriptor. Este DTO descriptor proveerá el nombre del mensaje que devuelve la clave funcional (no técnica ni de base de datos) del objeto del dominio que representa.

Por su parte, el ResultDescriptor describe el resultado de una operación, para lo cual especifica dos atributos:

- operationResultDescriptor: describe la naturaleza del resultado de la operación, de la misma forma en que se describe un parámetro. Es por esto que justamente se utiliza un ParameterDescriptor, delegándole a este último esta descripción.
- accessorDescriptors: es una lista, posiblemente nula, de formas de acceso al resultado obtenido (punto anterior), cada una de ellas descripta mediante un AccessorDescriptor. A continuación se amplía este punto.

Tal como se comentó en la explicación del modelo básico, muchas veces el interés no radica en el objeto obtenido como resultado sino en algún atributo que éste contenga. Puede extenderse esta idea logrando mayor profundidad: puede interesar el atributo de algún objeto que a su vez se constituya como atributo del objeto resultado obtenido, y así para mayores niveles. En todos estos casos, es necesario proveer una manera de acceder a al atributo de interés.

La especificación de cada forma de acceso al resultado se logra mediante un AccessorDescriptor. Éste especifica una secuencia de operaciones que se ejecutan en cascada, la primera de ellas sobre el resultado obtenido y posteriormente cada una de las restantes (si las hay) sobre el resultado de la anterior. Como cada uno de estos elementos no es nada más ni nada menos que una operación, estas operaciones de acceso se describen, cada una de ellas,

mediante un OperationDescriptor.

Es conveniente repetir que cada AccessorDescriptor describe una forma de acceso al resultado (la cual puede implementarse por medio de varias operaciones en cascada), pero en el contexto de uso de testing, el usuario deberá seleccionar sólo un AccessorDescriptor de todos los que se le ofrecen. Es decir, que el hecho que un ResultDescriptor defina varios AccessorDescriptor significa solamente que se ofrece al usuario una amplia gama de accesos, de las cuales, en cada ocasión, el usuario deberá optar solamente por una de ellas.

Cabe notar nuevamente que el usuario final no tendrá conocimiento de la secuencia de operaciones por las que está compuesto internamente un AccessorDescriptor, ya que no es de su interés ni pertinencia el conocimiento de la estructura interna de los objetos. El usuario solo verá un conjunto de ResultAccessors, cada uno con su descripción, y elegirá uno de ellos.

Cuando el usuario elija un AccessorDescriptor, internamente se tomará la secuencia de operaciones por las que está compuesto y éstas se setearán al OperationCommand como sus resultAccessorOperations (ver OperationCommand en el modelo básico).

6.2.4. Test Manager para Servicios

En la sección anterior se ha explicado cómo el usuario final puede operar con Descriptores de forma tal de manipular y configurar, a través de ellos, los elementos de la aplicación objetivo (servicios, operaciones, parámetros) que conformarán los tests que residen en la aplicación de testing.

En este punto cabe preguntarse quién es el responsable del modelo de mantener y gestionar estos descriptores. A primera vista esto le correspondería al TestManager ya introducido en el modelo básico. Sin embargo, se ha establecido como uno de los lineamientos de diseño que el modelo básico debe poder ser utilizado para una amplia gama de aplicaciones y fines. En el caso de que se asignara la responsabilidad de la gestión de descriptores de servicios al TestManager, se estaría violando este lineamiento, debido a que se estaría particularizando una clase de uso genérico para un fin determinado y en un contexto muy particular.

La solución propuesta viene dada nuevamente por una extensión mediante caja blanca. Esta consiste en extender la clase TestManager del modelo básico con una clase ServiceTestManager que implemente la gestión de descriptores de servicios. De esta manera, esta última clase podrá realizar todo lo que el TestManager puede realizar, más la gestión de descriptores. Asimismo, no hace falta modificar en absoluto a la clase TestManager, por lo que seguirá manteniendo toda la generalidad de uso establecida por el lineamiento previamente mencionado. En el caso de que se desee utilizar el modelo en el contexto de una aplicación objetivo tipo SOA, simplemente se deberá utilizar la clase ServiceTestManager en lugar de TestManager.

La figura siguiente muestra el modelo básico completo con la extensión para el uso en el contexto de servicios.

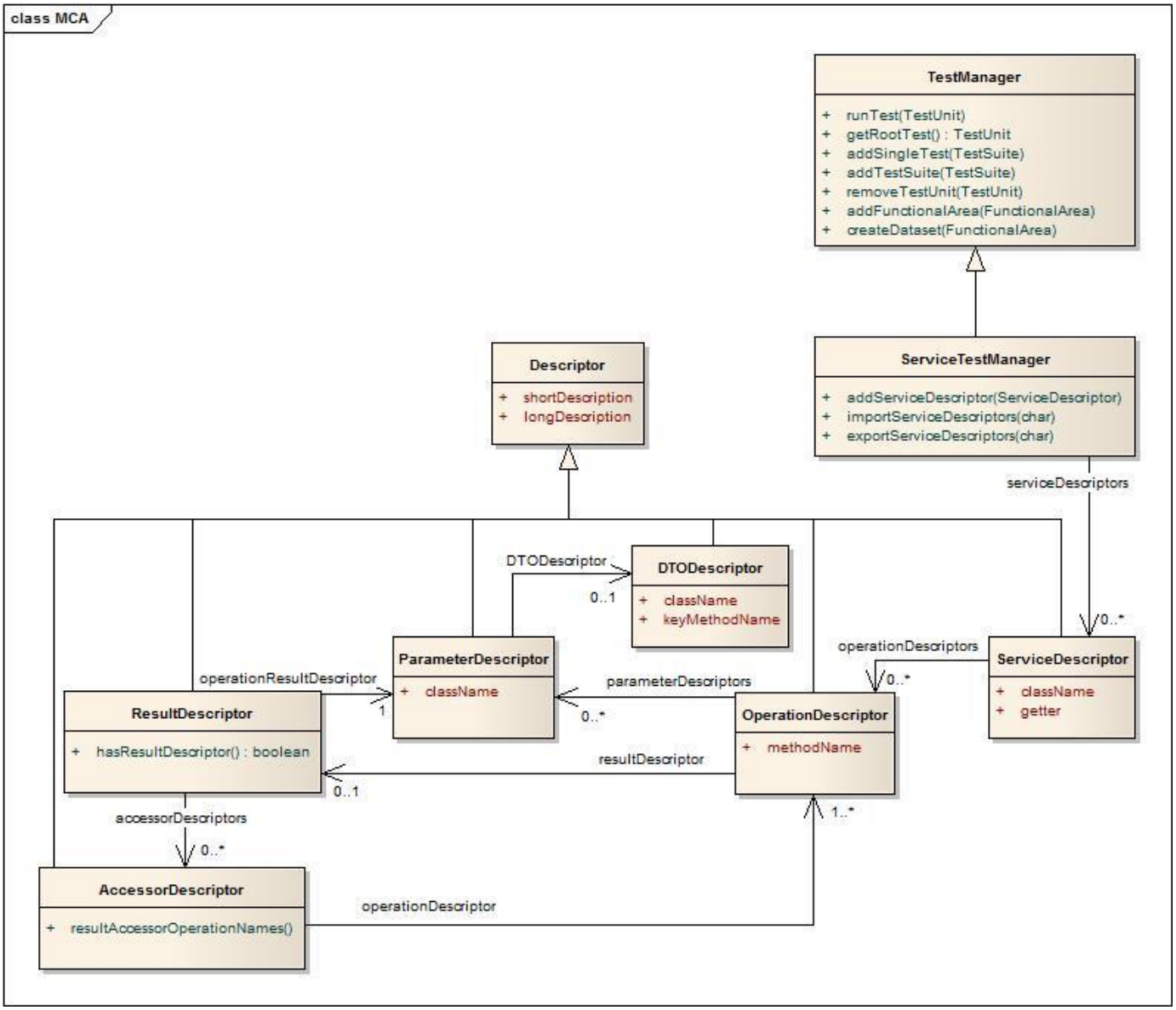


Figura 6.4 – Test Manager para Servicios

6.2.5. Importación de Descriptores de Servicios

En las secciones anteriores se describió el modelo de Descriptores de Servicios y el ServiceTestManager que los gestiona. Esto permite al usuario de la aplicación de testing disponer de una representación de los elementos existentes en la aplicación objetivo y que él puede utilizar para configurar los tests. Ahora bien: ¿cómo se alimenta este modelo de descriptores con la información concreta de los elementos de la aplicación objetivo?

Cabe destacar nuevamente el lineamiento de diseño sobre generalidad de uso, con lo cual el modelo de descriptores no debe contener, ni en su estructura ni en su funcionalidad, ninguna referencia a alguna aplicación objetivo en particular. La solución genérica propuesta en este trabajo consiste en la importación de tales descriptores mediante una estructura general en XML. De esta manera, los elementos de la aplicación objetivo que se desean importar como descriptores en la aplicación de testing son especificados mediante tal estructura XML. Por su parte, el ServiceTestManager ofrece una operación de importación de descriptores que toma como parámetro tal archivo XML y a partir de su información genera el modelo de descriptores en la aplicación de testing, dejando así todo preparado para que el usuario final de

tal aplicación pueda utilizarlos en la creación de tests.

A continuación se muestra una parte del archivo XML correspondiente a los ejemplos anteriores.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<servicesExport>
    <serviceDescriptors>
        <longDescription>Operaciones de Cuentas Escriturales</longDescription>
        <shortDescription>CuentaEscrituralServicio</shortDescription>
        <className>scut.services.CuentaEscrituralServicioImpl</className>
        <getter>getCuentaEscrituralServicio</getter>
        <operationDescriptors>
            <longDescription>Obtiene la Cuenta Escritural especificada</longDescription>
            <shortDescription>Obtener Cuenta Escritural</shortDescription>
            <methodName>getCuentaEscritural</methodName>
            <parameterDescriptors>
                <longDescription>Número de Cuenta Escritural asignado por la TGP</longDescription>
                <shortDescription>Número de Cuenta</shortDescription>
                <className>java.lang.String</className>
            </parameterDescriptors>
            <resultDescriptor>
                <longDescription>Cuenta Escritural de la TGP</longDescription>
                <shortDescription>Cuenta Escritural</shortDescription>
                <accessorDescriptors>
                    <longDescription>Número de Cuenta Escritural establecido por la TGP</longDescription>
                    <shortDescription>Número</shortDescription>
                    <operationDescriptors>
                        <longDescription>Número de la Cuenta Escritural establecido por la TGP</longDescription>
                        <shortDescription>Número</shortDescription>
                        <className>scut.dto.CuentaEscrituralDTO</className>
                        <methodName>getNumero</methodName>
                        <resultDescriptor>
                            <longDescription>Número de la Cuenta Escritural establecido por la TGP</longDescription>
                            <shortDescription>Número</shortDescription>
                            <operationResultDescriptor>
                                <longDescription>Número de la Cuenta Escritural establecido por la TGP</longDescription>
                                <shortDescription>Número</shortDescription>
                                <className>java.lang.String</className>
                            </operationResultDescriptor>
                        </resultDescriptor>
                    </operationDescriptors>
                </accessorDescriptors>
        </operationDescriptors>

                [....]
```

Figura 6.5 – Extracto de Descriptores de Servicios exportados a XML

Para la implementación particular realizada en este trabajo, la esta gestión de archivos XML en relación al modelo de objetos de descriptores se utiliza JAXB [JAXB]. Esto facilita la conversión automática de modelos de clases y objetos en representaciones XML y visceversa.

La confección del archivo XML a partir de los servicios y operaciones de la aplicación objetivo que se desean exportar puede realizarse manualmente o bien automáticamente mediante alguna herramienta de exportación confeccionada para tal fin. Siendo que los elementos a exportar corresponden a la aplicación objetivo y, por lo tanto, tendrán detalles y particularidades que le son propias en su contexto, se considera que no corresponde a la aplicación de testing implementar esta funcionalidad, exigiéndose solamente que el archivo XML exportado respete la estructura predefinida.

Sólo a los efectos de mostrar el funcionamiento del circuito completo en este trabajo, se ha implementado un proceso de exportación básico de los descriptores de los servicios de la aplicación objetivo desarrollada, la cual se describirá en detalle en secciones posteriores.

7. MODELO EXTENDIDO DE LA APLICACIÓN DE TESTING PARA PERSISTENCIA

En este capítulo se aborda el diseño de la solución para la persistencia de la Aplicación de Testing, y se describen de forma detallada todos los problemas inherentes a la compleja combinación de Reflection y Persistencia.

A continuación se realiza una explicación incremental del tema: primero se describe el problema que se necesita resolver, luego cómo se persisten objetos de forma genérica, seguidamente cómo se persiten métodos de forma genérica y, finalmente, cómo se integra esta persistencia de objetos y métodos con los componentes del modelo básico.

Las técnicas de extensión utilizadas aquí son las ya introducidas en el capítulo anterior.

7.1. Flexibilidad vs. Persistencia

La necesidad de adaptación del Modelo Básico a una gran variedad de contextos requiere un diseño con un muy alto grado de dinamismo y generalidad. Este objetivo se ha logrado, básicamente, combinando dos técnicas: tipado con clases genéricas y uso intensivo de reflection.

El tipado con clases genéricas implica que, por ejemplo, los parámetros y resultado de una operación sean tipados como Object, lo cual permite que cualquier objeto del sistema pueda ser utilizado como parámetro o devuelto como resultado.

Por otro lado, el uso intensivo de reflection implica, entre otras cosas, que la aplicación de testing no tiene programadas en su código fuente las operaciones de la aplicación objetivo que utilizará en sus tests, sino que las mismas se representan mediante objetos Method, cuya clase se define en el paquete de reflection de Java (java.reflect). La ejecución de estos métodos también se realiza con técnicas de reflection.

Estas dos técnicas (tipado genérico y reflection) funcionan perfectamente en el Modelo Básico hasta ahora descripto, alcanzando el grado de flexibilidad deseado. Si bien está ampliamente aceptado (arquitectura por capas) que al crear un modelo OO uno no debería pensar en cómo éste será persistido posteriormente, la realidad indica que muchas técnicas de persistencia no permiten persistir de forma directa "cualquier modelo OO" y además, por otro lado, Java define que existen objetos que no pueden serializarse.

En este trabajo la persistencia del modelo básico se realiza en una base de datos relacional mediante Hibernate, como mapeador O/R. Se encuentran así dos problemas a resolver: Hibernate no permite persistir la clase Object y Java define a Method como no serializable, por lo que tampoco puede persistirse.

Estos dos problemas derivan en que, si bien el modelo básico provee toda la funcionalidad que cubre las necesidades, se deben tomar algunos compromisos a la hora de persistirlo. Estos compromisos dependen de la tecnología de persistencia a utilizar. En esta sección se describen los compromisos de flexibilidad y generalidad que se asumen al persistir el modelo mediante Hibernate, proveyendo soluciones de diseño que permiten, aún así, cubrir las necesidades de creación, configuración y ejecución de tests de forma dinámica sobre aplicaciones objetivo de tipo SOA, tal como es el objetivo de este trabajo.

7.2. Dos operaciones básicas: Persistencia y Regeneración

En el modelo de persistencia que se describirá a continuación existen dos operaciones clave que se ejecutan sobre un objeto persistente, por lo que resulta conveniente definirlas de forma explícita; estas operaciones son la de persistencia y la de regeneración.

La Operación de Persistencia es aquella que permite descomponer un objeto del ambiente en varios fragmentos que puedan ser persistidos en una base de datos. Esto es bastante directo

en objetos que son serializables (y de hecho es la solución que emplea Hibernate), pero requiere de una solución ad-hoc para objetos que no lo son. Esto último, en este trabajo, ocurre con las clases Object y Method, dado el contexto altamente reflexivo inherente a los objetivos.

La Operación de Regeneración es la inversa de la operación anterior, y consisten en regenerar un objeto en el ambiente a partir de los fragmentos o representaciones del mismo que han sido persistidos mediante la Operación de Persistencia. Nuevamente, esto que es natural para objetos serializables, requiere de soluciones específicas para lo que no lo son.

7.3. Persistencia de Objetos

Tal como se ha explicado anteriormente, una de las maneras en que el modelo básico logra flexibilidad y versatilidad es mediante el tipado con clases genéricas, entre ellas Object. Esto permite que tanto parámetros como resultados pueda ser instancias de cualquier clase del sistema. No obstante esto, en el caso particular en que este modelo se persista con Hibernate, como es el caso de la aplicación de este trabajo, surge el problema de que la clase Object, tan genéricamente, no puede ser persistida. Es necesario definir estructuras de objetos más concretas, lo cual en principio atenta contra los lineamientos de generalidad propuestos en este trabajo.

La solución propuesta en este trabajo se basa en reducir la cantidad de clases que serán persistidas a unos pocos casos conocidos que, a su vez, cubran todas las necesidades de una aplicación SOA. En este sentido, hay en principio dos clases de parámetros y resultados que es seguro que podrán utilizarse en una aplicación objetivo implementada en Java, éstos son: Tipos Primitivos y Wrappers sobre Tipos Primitivos.

Además de estos valores y objetos de uso genérico, y tal como se ha mostrado en secciones anteriores, en muchas arquitecturas cliente/servidor y, en particular, en las aplicaciones SOA, es de amplia aplicación el patrón DTO (Data Transfer Object). Tal como su primera sigla lo indica, “Data”, estos objetos son los encargados de transferir información entre las aplicaciones cliente y el servidor. Esto significa que al utilizar DTOs, no es necesario que los clientes conozcan a los objetos del modelo del dominio, su estructura, ni la interrelación concreta entre los mismos.

En adición a lo anterior, en el caso de una arquitectura SOA, los clientes realizan el acceso al modelo no de forma directa, sino a través de una capa servicios. Estas dos estrategias combinadas, es decir, Servicios y DTOs, permiten un doble aislamiento del cliente con respecto al modelo de objetos subyacente. Este aislamiento permite a los clientes reducir la posibilidad de uso de innumerables clases de objetos del modelo a sólo algunas clases con estructuras mucho más simples: aquellas que, fuera del modelo, se constituyen como DTOs.

Con todo lo anterior, a continuación se describirá de qué manera puede mantenerse la funcionalidad del modelo básico persistiéndolo mediante Hibernate, reduciendo la persistencia genérica y completa de cualquier objeto a unos pocos y, eventualmente, de forma parcial. Se mostrará cómo pueden persistirse los tres valores/clases fundamentales y necesarios en una aplicación SOA:

1. Tipos Primitivos
2. Wrappers sobre Tipos Primitivos
3. DTOs

7.3.1. Persistencia de Tipos Primitivos

Si habláramos de un lenguaje de programación realmente orientado a objetos, esta sección

no tendría sentido, ya que en el contexto de tal paradigma se considera que "todo es un objeto". Sin embargo, Java define que existen ciertos "valores" (no objetos) que corresponden a ciertos "tipos" (no clases). En concreto, estos tipos primitivos definidos por Java son los siguientes: byte, short, int, long, float, double, boolean, char.

Estos valores son continuamente utilizados en todas las aplicaciones Java, por lo que su persistencia debe ser considerada. Las consecuencias de la violación del Paradigma Orientado a Objetos queda patente incluso dentro de la definición del mismo lenguaje. En este sentido, el programador necesita trabajar con estos valores (tipos primitivos) en lugares en donde sólo se admiten objetos y, tales valores, no lo son. Como ejemplo, tomemos el caso de la operación add(Object o) en una instancia de Collection. Como un valor primitivo no es un objeto, es imposible crear una Colection que los contenga. Ante esto, y para dar una solución al problema, Java define clases que actúan como Wrappers.

Una instancia (ahora sí un objeto) de un Wrapper (ahora sí una clase) simplemente tiene una variable que puede contener un valor primitivo. Tal es su función en el sistema. De esta manera, una instancia de un Wrapper, al ser un Object (por herencia) sí puede ser utilizada en lugares en donde se haya tipado con Object. Así, siguiendo el ejemplo anterior, si bien no se podrá crear una Colection de tipos primitivos, sí se podrá crear una colección con sus Wrappers.

En concreto existe una clase Wrapper para cada tipo primitivo, con lo cual Java define los siguientes wrappers para los tipos primitivos antes mencionados: Byte, Short, Integer, Long, Float, Double, Boolean, Char. Notar que en estos casos la letra inicial está en mayúscula, justamente porque representan clases.

Atento a esta necesidad de convertir ida y vuelta entre tipos primitivos y sus correspondientes wrappers, Java define en muchos puntos su conversión automática. El presente trabajo toma provecho de esta facilidad, trabajando muchas veces con los wrappers en lugar de con los tipos primitivos originales. Esto permite, en algunos puntos clave, unificar la forma de tratamiento considerando objetos a todos ellos.

En este sentido, si bien el modelo podrá trabajar tanto con tipos primitivos como con sus wrappers, a la hora de persistir uno de aquellos lo wrappeará y persistirá su correspondiente wrapper.

De esta manera, la persistencia de tipos primitivos se reduce a la persistencia de wrappers, la cual se especifica en la siguiente sección.

7.3.2. Persistencia de Wrappers

Tal como se desarrolló en la sección anterior, existen tantas clases wrappers como tipos primitivos, y cada instancia wrapper guarda un valor tipado estáticamente con el tipo primitivo que tal wrapper representa. Es decir, tal valor corresponde a un tipo primitivo.

Los valores de todos los tipos primitivos pueden convertirse a una representación de tipo String y, en sentido contrario, a partir de una representación de tipo String puede obtenerse tanto el valor del tipo primitivo original como el del wrapper que lo representa. Esta posibilidad de representar los valores de tipos primitivos mediante Strings, permite reducir la variedad de ocho representaciones originales diferentes (los tipos primitivos de Java) a una sola y única representación (String).

7.3.2.1. Operación de Persistencia

En línea con lo anterior, la persistencia de los diferentes wrappers correspondientes a los distintos tipos primitivos puede uniformizarse en un único método que consiste en persistir

siempre las representaciones textuales (Strings) del valor que contienen. Asociado a cada valor, además, debe persistirse el nombre de la clase del wrapper correspondiente, ya que será necesaria posteriormente para la regeneración de la instancia del wrapper original a partir de su representación textual.

Todas las clases Wrapper proveen mensajes polimórficos para la obtención de la representación textual del valor que contienen como así también para la obtención del nombre de su clase. Estos mensajes son: toString(), para el primer caso, y la secuencia getClass() + getName() para el segundo caso.

De esta forma, para cualquier wrapper w, sea cual fuere su clase concreta, puede obtenerse el par de Strings <value, className> necesario para su persistencia de la siguiente manera:

value = w.toSring();
className = w.getClass().getName();

Esta uniformidad permite que todas las instancias de wrappers puedan ser persistidas de la misma manera, tanto en cuanto a dinámica como en cuanto a estructura.

7.3.2.2. Operación de Regeneración

La regeneración contempla la obtención de una instancia de la clase wrapper que corresponda, a partir de la representación textual del wrapper, es decir a partir del par de Strings <value, className>.

Esta regeneración también puede uniformizarse para todas las clases wrappers utilizando reflection, mediante los siguientes pasos:

1. Obtener la clase a partir del nombre className.
2. Obtener el constructor de la clase obtenida en el punto anterior.
   Este constructor toma como parámetro un String, que es la representación textual del valor de tipo primitivo que el wrapper contendrá.
3. Crear una instancia a partir de la clase y constructor previamente obtenidos, tomando como parámetro la representación textual del valor: value.

Esta uniformidad de procedimiento permite que todas las instancias de wrappers puedan ser regeneradas de la misma manera, independientemente de su clase concreta.

7.3.2.3. Modelo de Wrappers Persistentes

En las dos secciones anteriores se describió cómo se realizan las operaciones de persistencia y regeneración de wrappers. Sin embargo, aún no se ha hecho explícita la estructura en la cual se persisten.

Ya se ha explicado que es deseable que los distintos wrappers sean persistidos en una misma estructura, y que el paso esencial hacia tal objetivo ha sido logrado a partir del momento en que todos ellos pueden ser reducidos a la representación textual <value, className>.

Ahora, tanto la estructura de este par como su tratamiento (operaciones de persistencia y regeneración) deben ser representados en nuestro modelo. Para esto se propone la clase "PersistentWrapper" que, como ya se dijo, estaría absorbiendo la persistencia tanto de tipos primitivos como de wrappers.

La estructura de la clase PersistentWrapper estaría entonces definida mediante dos variables de instancia, ambas de tipo String:

- value
- className

Esta estructura es simple, única y estática, por lo tanto puede persistirse en Hibernate de forma directa.

Por otro lado, el comportamiento de esta clase estaría dado básicamente por dos operaciones:

- setWrapper(Wrapper w)
- getWrapper()

La primera de ellas, setWrapper(Wrapper w), recibe como parámetro el objeto wrapper que se desea persistir y ejecuta la operación de persistencia tal como se describió anteriormente. Es decir: obtiene el par <value, className> que corresponde a w y con esos dos valores setea sus dos variables de instancia. De esta forma, mediante Hibernate, esta instancia de PersistentWrapper queda automáticamente almacenada en la Base de Datos.

La segunda operación, getWrapper(), implementa la operación de regeneración conforme a los pasos exactos que sea han descripto en la sección anterior, tomando como valores componentes del par a los valores de las variables de instancia value y className que, por medio de Hibernate, se encontraban transparentemente almacenados en la base de datos.

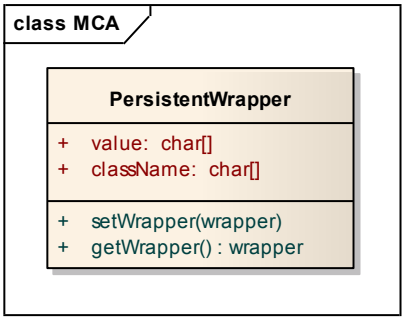


Figura 7.1 – Persistent Wrapper

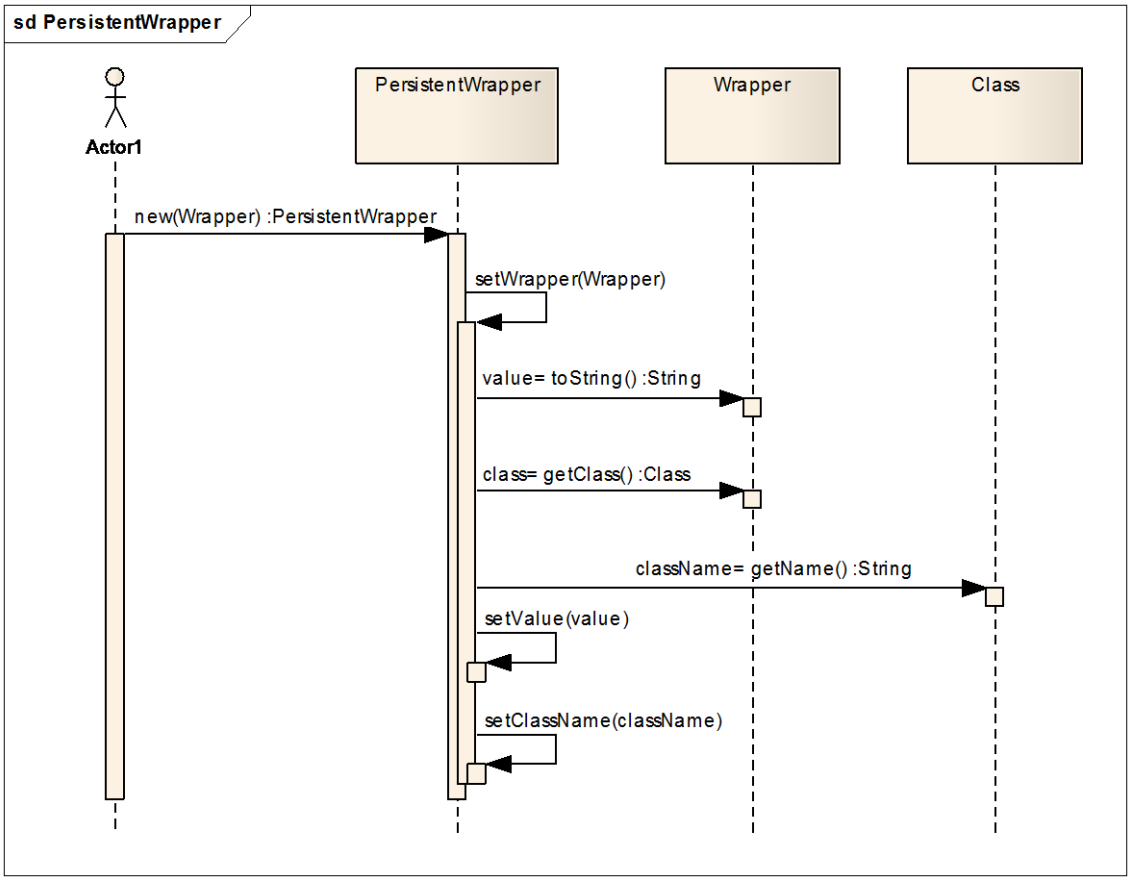


Figura 7.2 – PersistentWrapper: Operación de Persitencia

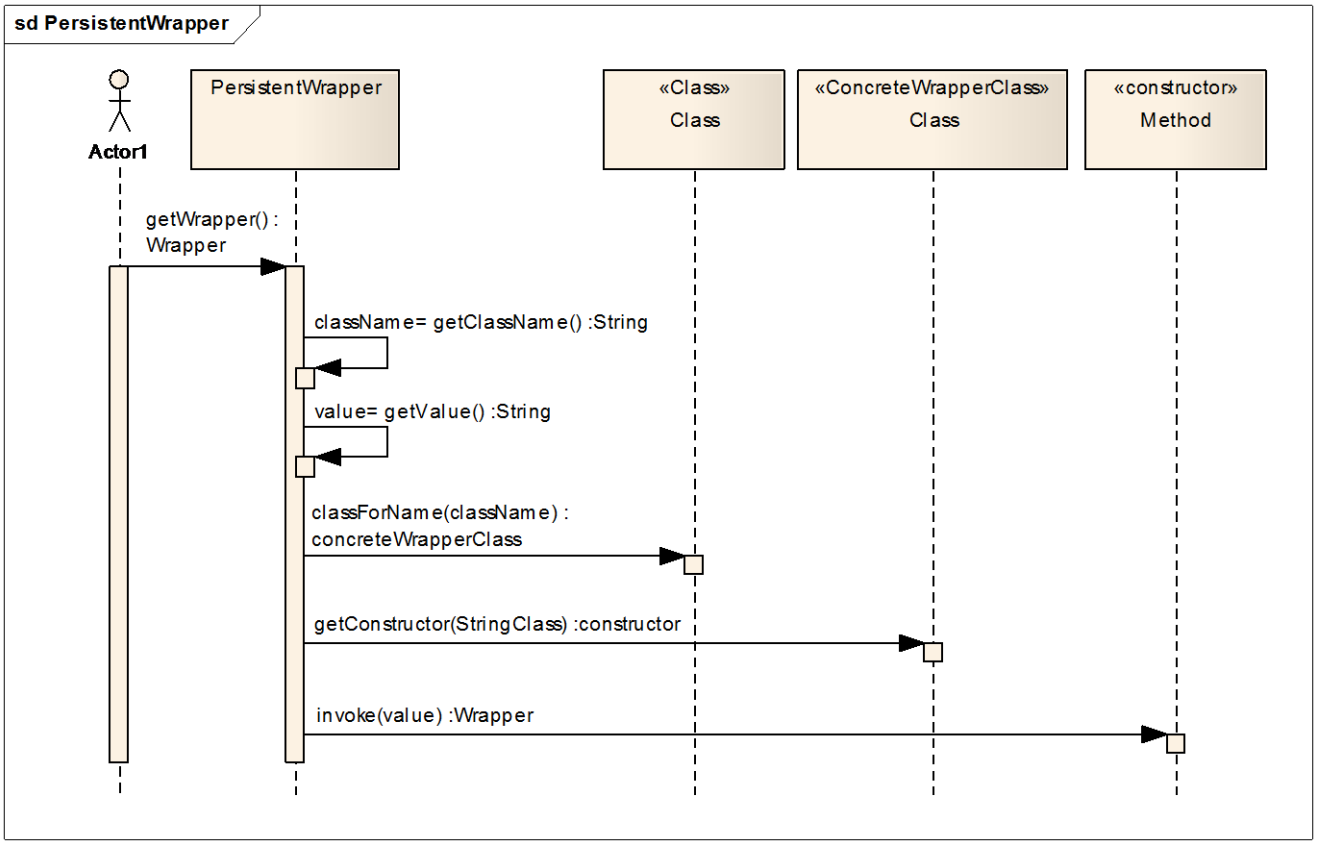


Figura 7.3 – PersistentWrapper: Operación de Regeneración

Con esta simple estructura y operaciones básicas, el modelo puede persistir y recuperar wrappers, lo cual también abarca la gestión de tipos primitivos. En la siguiente sección se describirá cómo esta clase evoluciona para abarcar también la persistencia de otro tipo de objetos.

7.3.3. Persistencia de DTOs

A diferencia de los wrappers y tipos primitivos, resulta más complejo encontrar una estructura homogénea que permita representar las múltiples variedades de DTOs que pueden encontrarse en las distintas aplicaciones objetivo a las que el modelo de testing debe adaptarse. Mientras conocemos exactamente cómo son los tipos primitivos y wrappers que provee Java, no conocemos de antemano cómo serán los DTOs que la aplicación objetivo proveerá.

En esta sección se estudia el uso genérico que se realiza de los DTOs en las aplicaciones de tipo SOA, y en base a esto se definen algunas estrategias para dar solución a esta demanda de persistencia y dinamismo.

7.3.3.1. Alternativas de diseño

Si bien no se conocen de antemano las estructuras de los DTOs de la aplicación objetivo antes de conocerla, estas estructuras sí se conocen con la máxima precisión en el momento en que se define cuál es la aplicación objetivo. Esto último permitiría definir la forma de persistencia de los DTOs de manera concreta en el momento de configuración de la aplicación de testing en relación a la aplicación objetivo.

En este sentido, el usuario configurador podría definir manualmente las estructuras de

persistencia residentes en la aplicación de testing de los DTOs definidos por la aplicación objetivo. Si bien esta alternativa es válida, la cantidad de tipos de DTOs que las aplicaciones definen suele ser tan grande que la especificación manual y explícita de sus formas de persistencia tornaría muy tediosa la tarea de configuración.

Una alternativa de solución podría ser la creación de una herramienta que, aprovechando el hecho conocido de que los DTOs tienen estructuras relativamente simples (por ejemplo, muy orientada a "propiedades"), genere automáticamente las estructuras de persistencia para cada DTO de la aplicación.

Estas alternativas podrían denominarse de "estructura estática" ya que, manual o automáticamente, lo que se consigue finalmente es una estructura de persistencia que se fija de forma definitiva y previa al momento de ejecución. Como contrapartida, puede pensarse en estrategias de persistencia más dinámicas, en donde no sea necesario la definición previa de ninguna estructura.

Se considera que ambas líneas de trabajo son válidas, ya que ninguna de ellas invade a la aplicación objetivo ni disminuye la generalidad del diseño y construcción de la aplicación de testing. Es decir que en ambos casos se continúan respetando los lineamientos de diseño definidos al principio de este trabajo.

En este trabajo se define una estrategia del segundo tipo, dinámica, lo cual implica que los DTOs se almacenan de forma automática, sólo cuando fuera necesario, de forma total o parcial según las necesidades y en tiempo de ejecución.

7.3.3.2. Usos de DTO

Para definir una estrategia adecuada en cuanto a la persistencia dinámica de DTOs conviene primero saber exactamente en qué casos esta es realmente necesaria y de qué modo realizarla. Como primer paso, a continuación se realiza una clasificación y análisis del uso de los DTOs en aplicaciones SOA.

El DTO se utiliza:

1. Como resultado de una operación

2. Como parámetro de entrada de una operación. En este caso, se distinguen dos situaciones:

    a. DTOs inmutables:

        El DTO que se envía como parámetro provino originalmente del modelo en una colección junto con otros DTOs, de forma tal de mostrar su información al usuario para guiarlo en la selección de uno de ellos. En este caso, el DTO que el usuario seleccione será enviado nuevamente a la capa de servicios como parámetro en la operación a ejecutar, pero sin modificarlo en absoluto. Es decir, el DTO vuelve como parámetro a la capa de servicios en su mismo estado original.

    b. DTOs editables:

        El DTO es editado en el cliente. Esto ocurre, por ejemplo, en las operaciones de edición de una entidad del modelo. En estos casos, el DTO que contiene la información original de la entidad es modificado en algunos de sus atributos, y a continuación el DTO modificado es enviado como parámetro a la capa de

servicios.

7.3.3.3. Estrategias de persistencia según el uso

En esta sección se describe el tratamiento de la forma de persistencia de un DTO según sea su uso. Esta persistencia podrá ser necesaria o no, o bien total o parcial (sólo algunos de sus atributos). Cabe destacar nuevamente que la estrategia de persistencia del modelo básico se ajusta, en nuestro caso, al contexto reducido de tecnología Hibernate con aplicaciones objetivo de tipo SOA.

En primer lugar conviene tener en cuenta que cuando un DTO representa una entidad del modelo, éste suele tener como atributo a la clave funcional de la misma. Esto permite que a partir de ella pueda recuperarse el resto de los atributos del DTO, lo cual ahorra la necesidad de almacenar un DTO completo cuando, según los usos descriptos en la sección anterior, el mismo es inmutable.

Caso 1: DTO como Resultado

En el caso de que el DTO se obtenga como resultado de una operación, se trata de un DTO inmutable, es decir, no ha sido modificado por la aplicación cliente. En este caso, bastaría con persistir el atributo que define la clave funcional de la entidad del modelo que el mismo representa. Por otro lado, no siempre es necesario persistir el DTO, sino que a veces puede persistirse sólo alguno de sus atributos. Esto ocurre, por ejemplo, cuando el usuario configura un test con una operación que devuelve un DTO para el que define, además, una forma de acceso a alguno de sus atributos (mediante un Accessor, como se explicó en secciones anteriores). En estos casos, el resultado final obtenido es el resultante de la aplicación del accessor, por lo que no es necesario persistir el DTO obtenido como resultado de la aplicación original.

En contrapartida, además de la persistencia del resultado final obtenido, debe también persistirse el resultado esperado contra el cual se comparará. En la especificación de este resultado esperado, un usuario nunca necesitará especificar un DTO completo, sino que especificará uno o varios atributos de la entidad subyacente. En todo caso, en el caso que desee realizar una aserción contra un DTO completo, podrá especificar la clave funcional de la entidad del modelo que éste representa.

Caso 2.a: DTO como parámetro de entrada inmutable

En este caso, sólo se requerirá la persistencia del DTO en el momento en que el usuario configurador de un test especifica a tal DTO como parámetro de una operación. Dado que el DTO es inmutable, bastará con persistir su clave funcional, ya que a partir de la misma la capa de servicios recuperará la entidad original (la cual, a su vez, determina el resto de los atributos que define el DTO).

Caso 2.b: DTO como parámetro de entrada editable

En este caso, los atributos del DTO no podrán ser recuperados a partir de su clave funcional, ya que los mismos pueden haber sido modificados con respecto al modelo de la aplicación objetivo.

Este es el caso más complejo, ya que el DTO no puede ser tratado como una unidad, sino que para su persistencia debe ser descompuesto en los atributos que contiene, o al menos en aquellos que han sido o han podido ser modificados.

En el contexto de testing, la persistencia de estos valores podrá ser necesaria sólo en la especificación de campos de un DTO cuando el mismo aparezca como parámetro de una

operación en la configuración de un test. Nuevamente aquí podremos generalizar este escenario de persistencia mediante la reducción a una estructura fija pero genérica. En concreto, en este escenario, podrán persistirse sólo aquellos atributos que se consideren necesarios (y no obligatoriamente todos ellos) mediante pares <nombreAtributo, valor>. En donde el campo valor define, a su vez, alguno de los elementos persistibles vistos hasta ahora: tipo primitivo, wrapper o DTO (definición recursiva).

7.3.3.4. Persistent Object

Hasta aquí se han descripto técnicas que permiten persistir tipos primitivos, wrappers y DTOs. Se ha descripto una clase (PersistentWrapper) que permite persistir de forma unificada a los dos primeros, por lo que resta resolver mediante qué técnica concreta se persistirán los DTOs.

En este punto es necesario remitirse al modelo básico para notar nuevamente que el diseño del mismo es genérico, y en esta línea los parámetros y resultados de las operaciones (entre otras cosas) están tipados como Object. Es por esto que, a pesar de que ya en este punto se sepa que en este contexto reducido (Hibernate, SOA) alcanza con persistir tres tipos de elementos (tipos primitivos, wrappers y DTOs) es aún deseable que la persistencia de ellos se lleve a cabo de una forma unificada y bajo una misma estructura. Esto permitirá tratar con "Objetos Persistibles", independientemente de cuál de los tres tipos se trate.

Las ventajas de este tratamiento homogéneo y transparente se verán claramente en las siguientes secciones, cuando se integren finalmente estas estructuras al modelo básico, extendiéndolo.

Para lograr este objetivo, se redefinirá levemente la clase "PersistentWrapper" (que ya integra la persistencia de tipos primitivos y wrappers) de forma de que logre abarcar también la persistencia del tercer tipo de elemento que debe persistirse: DTO. De esta forma, pudiendo persistir todos los tipos de objetos que se necesitarán en este contexto reducido, se puede decir que esta clase extendida representa ahora "Objetos Persistentes", por lo que su nombre final se define como "PersistentObject".

De la misma manera que en su versión anterior (PersistentWrapper), a continuación se explica la estructura y comportamiento de la clase PersistentObject, que varían muy poco en esta nueva versión con respecto a la anterior.

Nuevamente nos remitiremos a los tipos de usos de DTOs para comentar cómo se logra su persistencia mediante esta nueva estructura (PersistentObject), que también será persistida mediante Hibernate.

Casos 1 y 2.a: DTOs inmutables

Como ya se ha descripto, los casos 1 y 2.a tratan de DTOs inmutables, sea que se obtengan como resultado o que se utilicen como parámetro de una operación. En estos casos, como ya se ha visto, bastará con persistir la clave funcional de la entidad que el DTO representa.

Esto significa que también puede representarse mediante la estructura textual <value, className>, siendo ahora, para DTOs inmutables, la primera componente la clave funcional del mismo y la segunda el nombre de su clase. Se ve que la estructura de persistencia de wrappers y DTOs es la misma, por lo que se ratifica y mantiene para la clase PersistentObject la estructura ya definida en secciones anteriores, es decir, dos variables de instancia:

- value: representación textual del valor (si es un wrapper) o clave funcional (si es un DTO).
- className: nombre de la clase a la que corresponde el valor; es decir, la clase concreta

del wrapper o del DTO, según lo que represente.

Habiendo abarcado mediante una única estructura, hasta ahora, casi todos los objetos que se necesitan persistir (tipos primitivos, wrappers y DTOs inmutables), se describirá a continuación la parte dinámica de la persistencia, es decir las dos operaciones básicas que el PersistentObject debe implementar: operación de persistencia y operación de regeneración.

7.3.3.4.1. Operación de Persistencia (para los casos 1 y 2.a)

De forma similar a como se describió en secciones anteriores, PersistentObject define la operación de persistencia mediante el mensaje setObject(Object o). Este mensaje recibe el objeto a persistir, que ahora además de un wrapper también puede ser un DTOs inmutable, y a partir del mismo obtiene el par <value, className> con cuyos valores setea sus variables de instancia descriptas arriba. De esta manera, por estar persistido mediante Hibernate, el PersistentObject queda almacenado automáticamente en la base de datos.

Ahora bien, en las explicación de la sección anterior, en la cual sólo se persistían wrappers (PersistentWrapper), se había mostrado que la operación de persistencia se reducía a las siguientes dos operaciones más básicas:

value = w.toSring(); (1)
className = w.getClass().getName(); (2)

Aquí conocíamos la naturaleza de w, sabíamos que era una instancia de un wrapper aunque no supiéramos exactamente de cuál. Sin embargo, gracias al polimorfismo, sabíamos que cualquiera que fuere la clase de la instancia, ésta entendería el mensaje toString(), posibilitándonos obtener de manera homogénea la representación textual del valor (value). De esta forma, la línea (1), quedaba representada de forma genérica como se muestra arriba.

En el caso de DTOs inmutables la representación textual del valor (value) que necesitamos obtener es la clave funcional del mismo. Sin embargo, la generalidad antes descripta en principio se pierde al intentar extenderla también a la persistencia de estos DTOs, ya que el objeto recibido ahora podrá ser también un DTO y éstos pertenecen a la capa de servicios de la aplicación objetivo, por lo que no se sabe qué es lo que devuelve el mensaje toString() (siendo que se necesita obtener la clave funcional).

Desde ya se descarta como alternativa de solución la posibilidad de redifinir el mensaje toString() de los DTOs de la aplicación objetivo, por resultar absolutamente invasiva. Por otro lado, la clave funcional de cada DTO se obtiene mediante un mensaje distinto, por lo que ni siquiera existe uniformidad de esta operación en la aplicación objetivo y, también para evitar una solución invasiva, no puede exigírsele a tal aplicación la definición de un mensaje genérico para la obtención de la clave en todos sus DTOs.

Debido a las razones anteriores, se propone una readecuación de la línea (1) anterior, de manera tal que pueda abarcar esta nueva necesidad de persistencia de una manera genérica y no invasiva. La forma propuesta para lograr esto consiste como parte de los descriptores de servicios ya descriptos se especifique también, para cada DTO en particular, el nombre del mensaje que devuelve su clave funcional. Luego, en tiempo de ejecución, y nuevamente mediante reflection, la aplicación de testing puede enviar al DTO recibido el mensaje con el nombre especificado. De esta manera, para el caso particular de un DTO, la línea (1) debería refactorizarse de la siguiente manera:

value = dto.invoke(keyGetterName);

En donde:

- dto: es la instancia de un DTO recibido.

- keyGetterName: es el nombre (textual) del mensaje implementado por la instancia recibida que devuelve su clave funcional. Este nombre es el que se especifica en los descriptores de servicios (DTODescriptor.keyMethodName).
- invoke: es el mensaje que provee Java como parte de su funcionalidad de reflection que permite que una instancia ejecute un mensaje del cual se tendrá su nombre concreto sólo en tiempo de ejecución (y no en tiempo de compilación), tal como es el caso presente.

De esta manera se logra la operación de persistencia de DTOs mediante las dos operaciones siguientes:

```
value = dto.invoke(keyGetterName); (1')
className = dto.getClass().getName(); (2)
```

Tal como puede apreciarse, la línea (2) no necesita ser modificada con respecto a la original para wrappers, ya que los mensajes que contiene son tan genéricos que permiten aplicarse a cualquier instancia. Es por esto que pueden ser entendidos tanto por wrappers como por DTOs.

Hasta aquí se tiene que se pueden persistir tanto wrappers como DTOs, solamente con la variación mencionada en (1) y (1'), de esta forma, el procedimiento genérico que recibe y persiste a cualquiera de ellos recibe tiene a su parámetro tipado como Object, lo cual permite persistir a estas dos clases de objetos y aún queda abierto a otras posibilidades que se surjan en el futuro. La implementación del mensaje setObject(Object o) de la clase PersistentObject quedaría entonces de la siguiente manera:

```
If (o.esWrapper){
        // o es un wrapper.
        value = w.toSring(); (1)
}else{
        // o es un DTO.
        value = dto.invoke(keyGetterName); (1')
}
className = w.getClass().getName(); (2)
```

De esta forma, la instancia de PersistentObject queda configurada como corresponde en sus atributos value y className, y automáticamente persistida por Hibernate. Posteriormente se describirá cómo se realiza la operación de regeneración.

Si bien el pseudocódigo mostrado arriba ya resuelve el problema de persistencia, el mismo muestra estar muy acoplado a los tipos de objetos que puede persistir (wrappers y dtos). Esto podría generalizarse aún más, permitiendo que la clase PersistentObject no se limite sólo a estas clases de objetos y quede abierta a la posibilidad de persistir otras clases de objetos que puedan aparecer en el futuro.

En este sentido, el punto a generalizar radica en la forma de obtención del valor textual del objeto a persistir, lo cual podría considerarse una forma de serialización. La generalización de este aspecto puede lograrse mediante la aplicación del patrón de diseño Strategy. De esta forma, un PersistentObject podrá delegar la responsabilidad de serialización del valor en una de dos posibles estrategias configuradas dinámicamente en el sistema.

En el modelo extendido estas estrategias se representan mediante la clase SerializationStrategy y su subclase ServiceSerializationStrategy; la primera de ellas permite serializar (para su inmediata posterior persistencia) wrappers, mientras que la segunda agrega la capacidad de serializar DTOs. El mensaje polimórfico que ambas implementan es serializeValue(Object o) y devuleve un String que es la representación textual de o. A este mismo mensaje, las clases SerializationStrategy y ServiceSerializationStrategy lo

implementarán de distinta manera: la primera tal como se describió en (1), mientras que la segunda podrá alternar entre (1') y (1), esto último heredando comportamiento de su superclase.

Con esta solución de diseño, el PersistentObject ya no tiene rastros de los objetos concretos que puede persistir ya que delega polimórfica y transparentemente en la estrategia de serialización que el usuario configurador haya definido dinámicamente en el sistema. El pseudocódigo quedaría de la siguiente manera:

value = serializationStrategy.serializeValue(o); *(1 y 1' homogeneizados)*
className = w.getClass().getName(); *(2)*

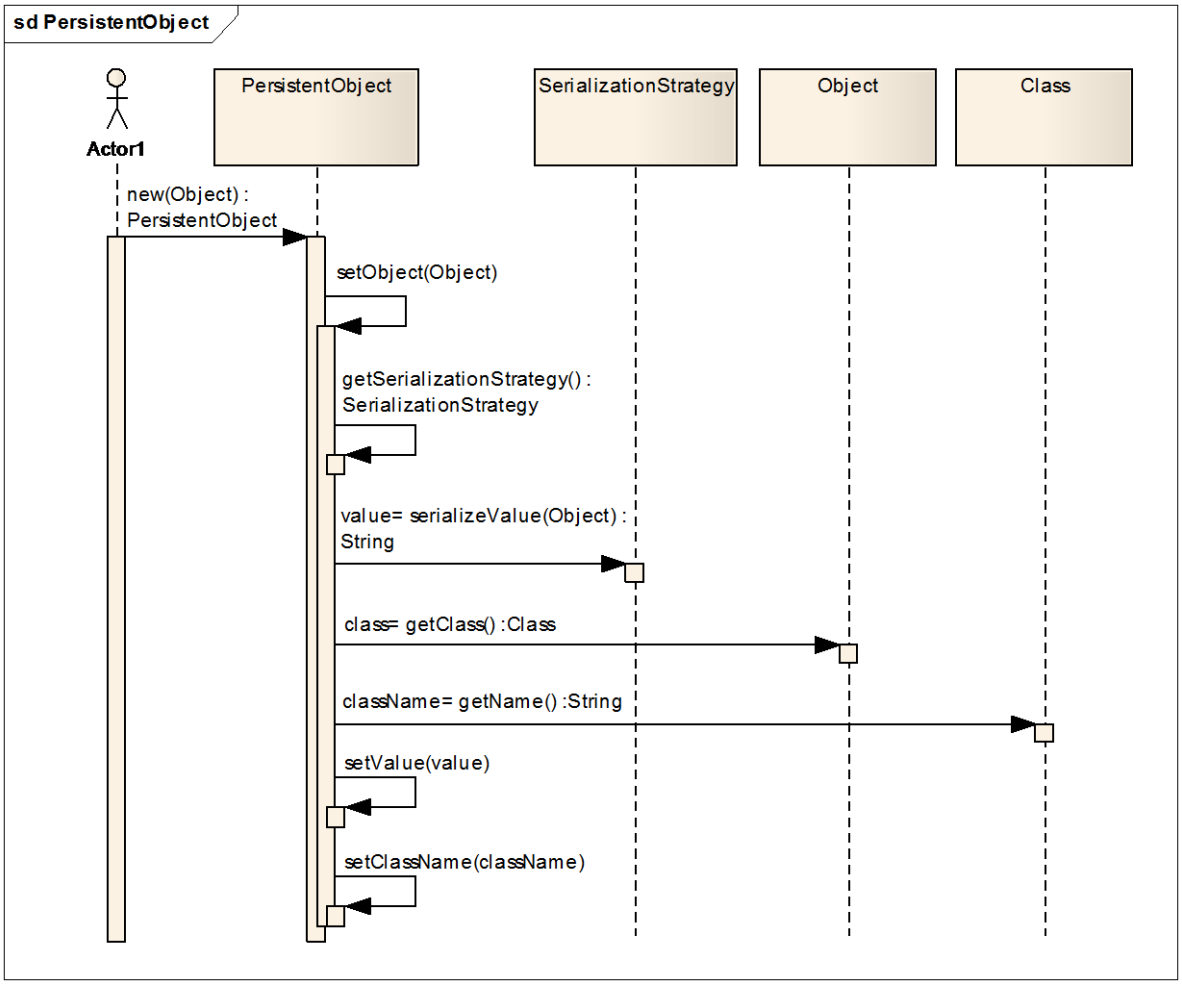


Figura 7.4 – PersistentObject: Operación de Persistencia

Así, PersistentObject permite que todas las instancias de wrappers y DTOs inmutables puedan ser persistidas de la misma manera, tanto en cuanto a dinámica como en cuanto a estructura.

7.3.3.4.2. Operación de Regeneración (para los casos 1 y 2.a)

La operación de regeneración de la clase PersistentObject se llama getObject(), y permite obtener un objeto a partir de los atributos textuales value y className definidos en tal clase.

En el caso en que la aplicación objetivo defina para cada DTO un constructor que reciba como parámetro un String con su clave funcional, entonces la implementación de la operación

de regeneración provista para los wrappers (clase PersistentWrapper), podría utilizarse de forma directa, sin ningún tipo de modificación. Aplicada al caso de los dtos, se seguiría utilizando de la misma manera que la ya descripta:

1. Obtener la clase a partir del nombre className (que ahora es el nombre de la clase del DTO).
2. Obtener el constructor de la clase obtenida en el punto anterior (que ahora es el constructor del DTO). Este constructor toma como parámetro un String, que ahora es la representación textual de la clave funcional del DTO.
3. Crear una instancia a partir de la clase y constructor previamente obtenidos, tomando como parámetro la representación textual de la clave funcional: value.

Si bien, como se ve, se podría reutilizar la operación de regeneración original (la ya utilizada para wrappers) sin ningún tipo de modificación para extenderla al uso con DTOs, se debería exigir a la aplicación destino que los DTOs implementen este tipo particular de constructor. Esto último, de acuerdo a los lineamientos de diseño que se siguen en este trabajo, se considera una exigencia invasiva en relación a la aplicación destino, puesto que condiciona su construcción o directamente impide su testeo.

Nuevamente se busca una alternativa de solución de diseño no invasiva con respecto a la aplicación objetivo, y se consigue de una manera similar a las soluciones flexibles alcanzadas hasta ahora: se permite que la forma de creación del DTO sea configurada externamente (por ejemplo en un archivo xml), y en tiempo de ejecución la creación del DTO se lleva a cabo mediante reflection, consumiendo tales valores previamente especificados.

Se debe permitir que el usuario especifique, para cada DTO, una de dos posibles maneras de crearlo:

A. Mediante un constructor con un parámetro que representa su clave funcional (tal como es el caso descripto arriba).
B. Mediante un constructor vacío y un método setter que, inmediatamente luego de creada, permita setear a la nueva instancia de DTO la clave funcional.

La especificación de estos valores, que pueden realizarse en un archivo xml, finalmente alimentarán el modelo de descriptores de forma que el modelo de testing pueda tomarlos en el momento de ejecutar la operación de regeneración.

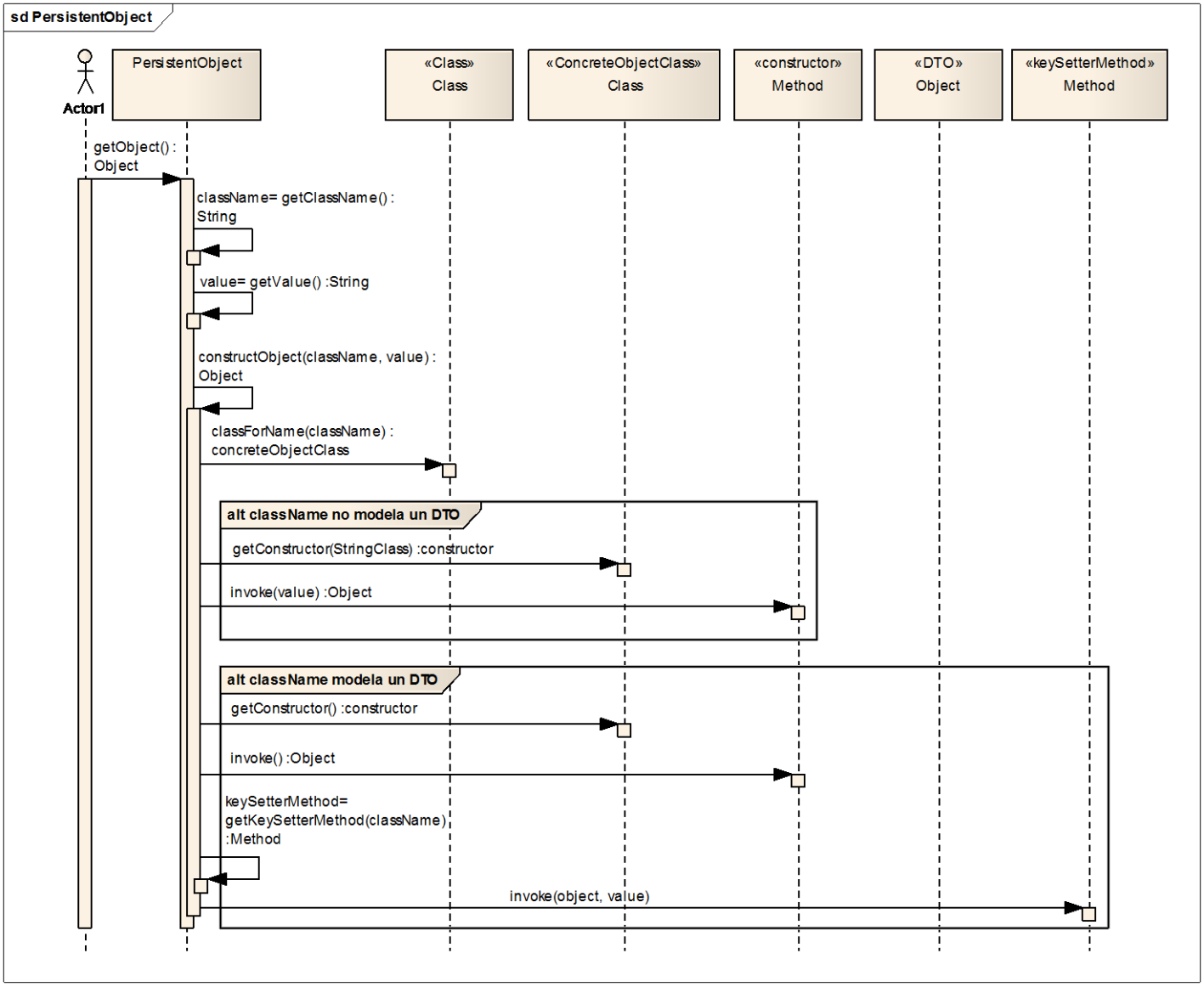


Figura 7.5 - PersistentObject: Operación de Regeneración

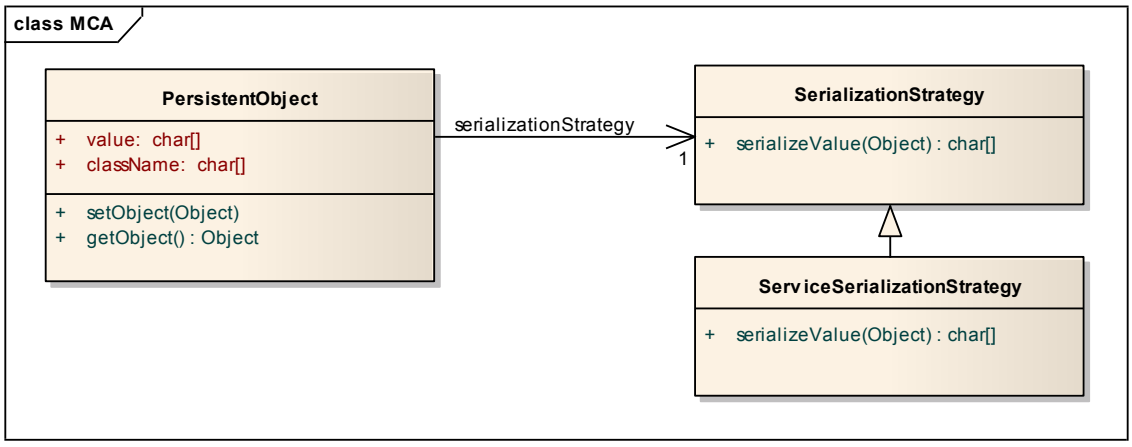


Figura 7.6 – Estrategias de Serialización

Caso 2.b: DTO como parámetro de entrada editable

En esta sección se describe un paso más en la evolución del modelo de persistencia que se viene desarrollando, ahora extendiendo el mismo (en particular, a la clase PersistentObject) de forma de que implemente la persistencia de los DTOs cuyos campos pueden ser editados.

En cuanto a la estructura, el modelo de DTO editable utiliza misma estructura provista por PersistentObject y, además, agrega un elemento estructural nuevo: el conjunto de propiedades (atributos) que pueden ser persistidos. Cada una de estas propiedades se define mediante un par <propertyName, value>, en donde la primera es el nombre del atributo (a partir del cual automáticamente podrán generarse getters y setters) y el valor textual del mismo.

Es este reúso de la estructura lo que, en principio, justifica el hecho de crear una nueva clase, subclase de PersistentObject, que, además, agregue la estructura correspondiente a la gestión de propiedades. De esta manera, se crea la nueva clase PersistentDTO, quedando el modelo de la siguiente manera:

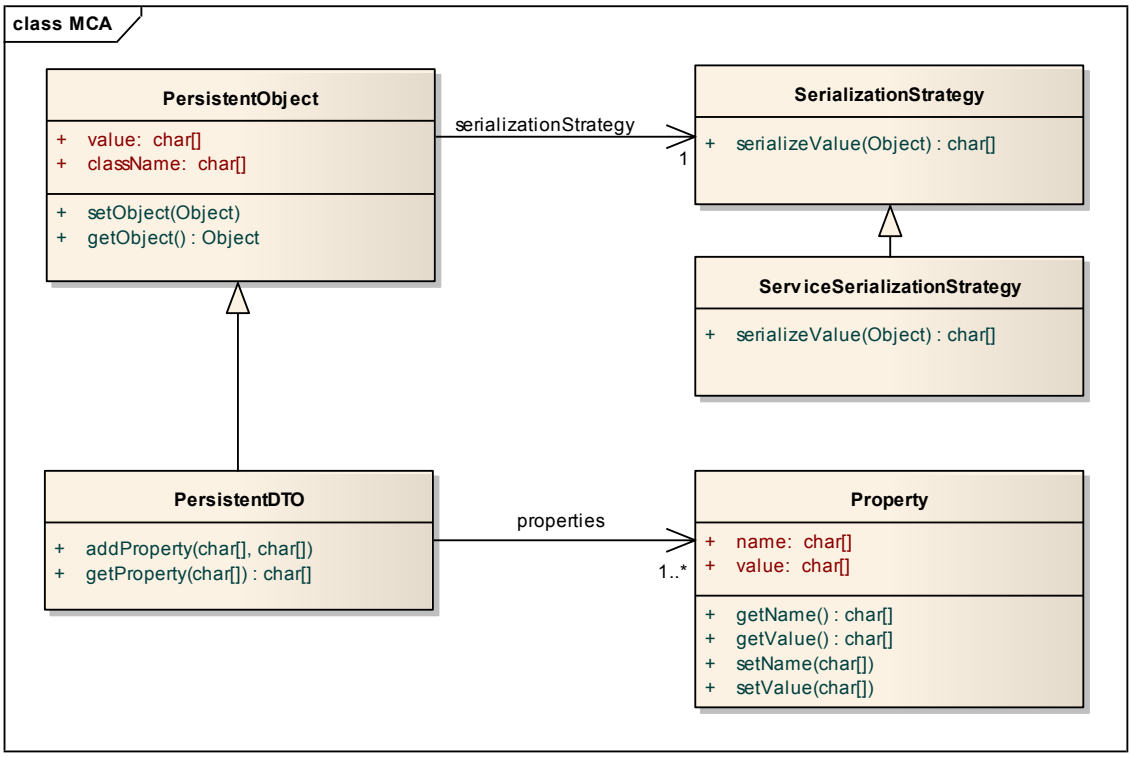


Figura 7.7 – PersistentDTO como parámetro de entrada editable

Esta clase se utilizaría como una extensión a ser utilizada sólo en el contexto en que se utilice persistencia, SOA y existan DTOs editables, en caso contrario no sería necesario su uso. De esta manera, los contextos en que deben utilizarse estas clases de persistencia son:

- PersistentObject: esta clase basta por sí sola si la aplicación objetivo sólo gestiona tipos primitivos, wrappers y/o DTOs inmutables.

- PersistentDTO: debe considerarse la extensión del modelo provista por esta clase si los servicios de la aplicación objetivo que se desean testear (pues podemos decidir sobre qué servicios nos enfocaremos durante el testing) gestionan DTOs editables.

Habiendo definido su estructura, la parte dinámica incluye las operaciones de persistencia y regeneración, que al igual que en su superclase se definen mediante los mensajes setObject(Object o) y getObject(), respectivamente.

En la clase PersistentDTO, ambas operaciones re-utilizan las implementaciones que ya existen en su superclase, lo cual termina de justificar la relación de herencia. Este re-uso se explica porque la superclase PersistentObject implementa lo referente a la persistencia del DTO en sí, mientras que la subclase PersistentDTO amplía este comportamiento implementando la persistencia de sus propiedades.

A su vez, estas propiedades podrán ser tipos primitivos, wrappers, DTOs inmutables o DTOs editables, por lo que, mediante una definición recursiva, quedan cubiertos todos los casos de persistencia de objetos que se necesitan en el contexto de nuestro interés, o sea: una aplicación de testing persistida mediante Hibernate configurada para una aplicación objetivo con arquitectura SOA.

7.4. Persistencia de Métodos

La persistencia de métodos es bastante más simple que la de objetos descripta hasta ahora. Esto se debe a que si bien no se conoce de antemano las clases ni estructuras concreta de los objetos que se utilizarán, sí se conoce perfectamente la composición de los objetos Method, ya que son los definidos por Java para reflection.

La versión persistente de un java.reflect.Method será representada en el modelo extendido mediante la clase PersistentMethod. De la misma manera que los PersistentObjects, esta clase definirá una estructura y las operaciones de persistencia y regeneración.

La estructura consta de algunos pocos atributos textuales, con lo cual esta clase puede persistirse en Hibernate de forma directa. Estos atributos son:

- name
- parameterClassNames
- declaringClassName

Estos atributos definen el nombre del método, la colección de los nombres de las clases de los parámetros y el nombre de la clase que define al método en cuestión.

Mediante la descomposición de un java.reflect.Method en estos tres atributos, puede persistirse y regenerarse el mismo mediante las operaciones que se describen a continuación.

La clase PersistentMethod define dos mensajes, setMethod(Method m) y getMethod(), que implementan las operaciones de persistencia y regeneración, respectivamente. La implementación de estas operaciones, salvo algunas consideraciones de orden técnico de bajo nivel, resulta de implementación directa mediante reflection.

La persistencia mediante la operación setMethod(Method m), se logra simplemente pidiendo a al método m, los atributos que se necesitan para completar la estructura de la instancia de la clase PersistentMethod que lo representará.

Por otro lado, la regeneración mediante la operación getMethod(), se logra también de forma directa mediante reflection de la siguiente manera:

1. Se obtiene la clase que implementa el método, a partir del atributo declaringClassName.
2. Se obtiene la colección de clases de los parámetros, a partir de los nombres de clases contenidos en la colección parameterClassNames.
3. A la clase obtenida en el punto 1 se le pide el método cuyo nombre es name y las clases de sus parámetros son las obtenidas en el punto 2.

A continuación se muestra el modelo de la clase PersistentMethod y el extracto del código

Java que la implementa.

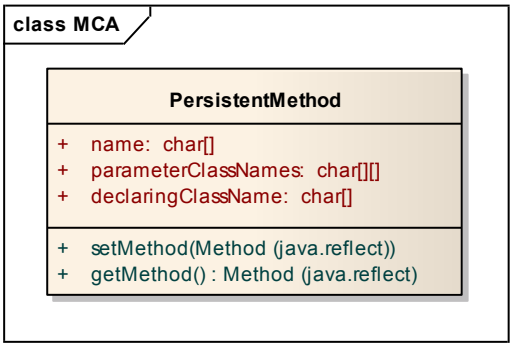


Figura 7.8 - PersistentMethod

En la sección siguiente se describirá cómo esta clase se integra al modelo extendido.

7.5. PersistentObjects y PersistentMethods en contexto

En las secciones anteriores se ha descripto cómo pueden persistirse objetos y métodos mediante PersistentObjects y PersistentMethods. Sin embargo, se ha explicado cómo se realiza la persistencia de los mismos de forma aislada, es decir, fuera del contexto del modelo de testing. En esta sección se detallará la forma en que los PersistentObjects y PersistentMethods se integran al modelo de testing, junto al resto de las componentes del mismo.

En primer lugar debe notarse que hay clases del Modelo Básico que conocen a instancias de las clases Object y/o de la clase Method, a saber: OperationCommand, SingleTestAssertion y AssertionStrategy. Estas clases no pueden ser persistidas de forma directa ya que algunos de sus atributos, aquellos de tipo Object y Method, no pueden persistirse de forma directa (por no ser serializables o no conocerse su estructura de antemano). Sin embargo, para poder persistir elementos de clase Object y Method se han creado las clases PersistentObject y PersistentMethod, respectivamente, tal como se ha descripto en las secciones anteriores.

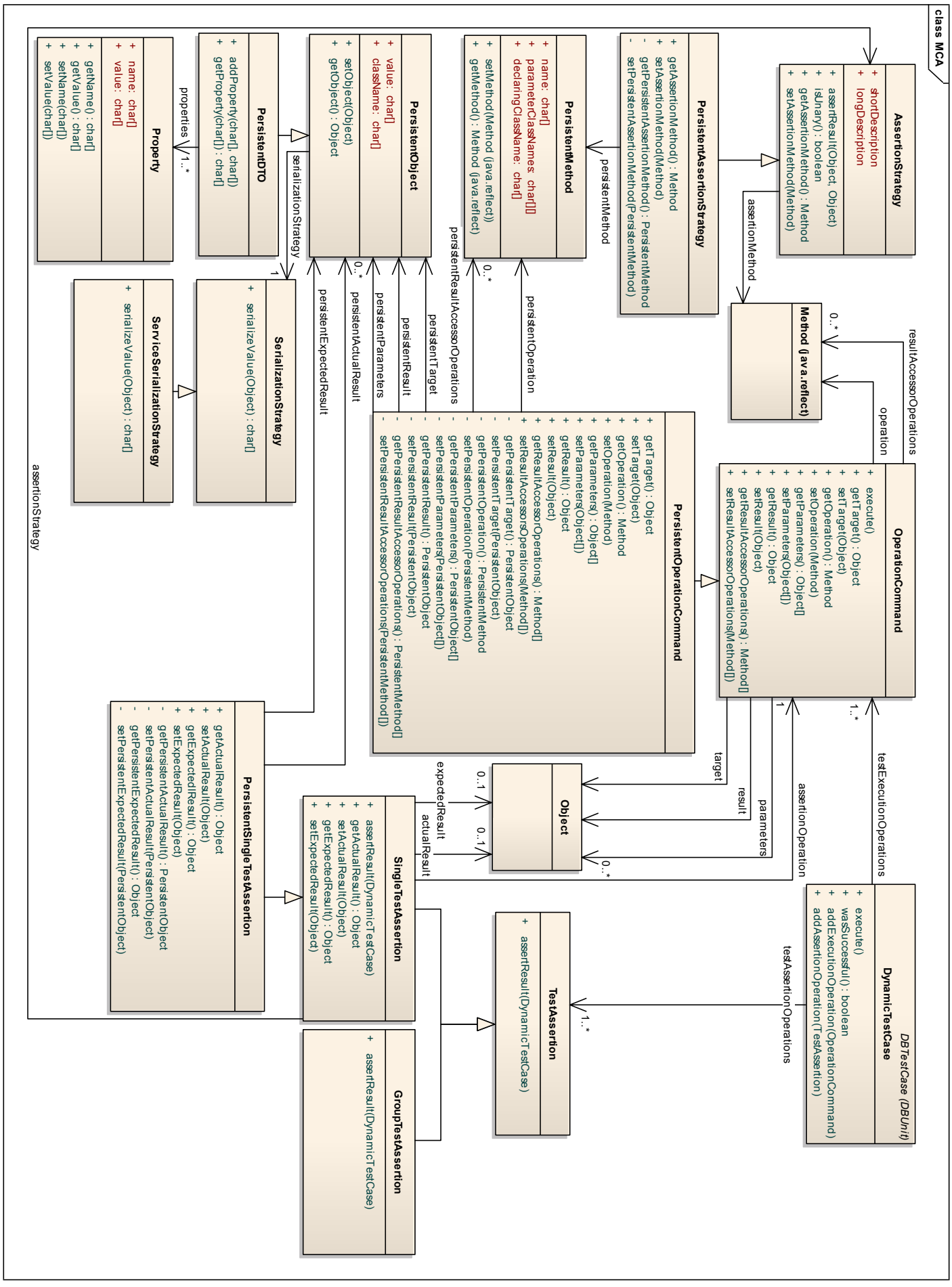
La cuestión que surge ahora es: ¿cómo integrar las clases PersistentObject y PersistentMethod al modelo básico para poder persistirlo pero sin invadirlo?

Cabe recordar que el modelo básico se plantea para un uso absolutamente genérico, independientemente de si éste es persistido o no y/o de la estrategia de persistencia a utilizar. En contrapartida, al extender el modelo para su adaptación a una persistencia mediante Hibernate, se han generado las clases PersistentObject y PersistentMethod, exclusivamente para este nuevo contexto reducido. Seguramente técnicas de persistencia OO más avanzadas puedan persistir el modelo básico de forma directa, sin necesidad de apoyarse en estas dos clases. Los lineamientos de generalidad establecen que el modelo básico debe quedar limpio y abierto a estas otras alternativas de persistencia, y es por esto mismo que no puede ser contaminado incluyendo clases que sólo cobren sentido en un contexto reducido y particular (Hibernate y SOA). De todo esto, se concluye que el modelo básico no debe hacer referencia directa a las clases PersistentObject y PersistentMethod. Entonces, ¿cómo incluir estas clases en el modelo?

La solución de diseño propuesta en este trabajo consiste en una extensión del modelo básico mediante caja blanca, es decir mediante herencia. Algunas clases del modelo básico serán extendidas con subclases que, a su estructura y comportamiento original, agregarán las pocas estructuras y comportamientos necesarios para gestionar la persistencia. Durante la ejecución, las instancias de estas subclases formarán parte de la dinámica del modelo, pero

gracias al principio de sustitución de Liskov el modelo las tratará con el mismo protocolo de su superclase, sin implementar ningún tratamiento nuevo ni distintivo.

En concreto, pueden persistirse de manera directa y natural todas las clases del modelo básico, a excepción de OperationCommand, SingleTestAssertion y AssertionStrategy, ya que conocen instancias de Object y Method, y éstos no pueden ser persistidos. Para el caso de estas tres clases, se crean tres subclases: PersistentOperationCommand, PersistentSingleTestAssertion y PersistentAssertionStrategy, tal como se muestra en la siguiente figura.


class MCA
AssertionStrategy
+ shortDescription
+ longDescription
+ assertResult(Object, Object)
+ isUnary() : boolean
+ getAssertionMethod() : Method
+ setAssertionMethod(Method)
assertionMethod
Method (java.reflect)
resultAccessorOperations
operation
OperationCommand
+ execute()
+ getTarget() : Object
+ setTarget(Object)
+ getOperation() : Method
+ setOperation(Method)
+ getParameters() : Object[]
+ setParameters(Object[])
+ getResult() : Object
+ setResult(Object)
+ getResultAccessorOperations() : Method[]
+ setResultAccessorOperations(Method[])
testExecutionOperations
DynamicTestCase
DBTestCase (DBUnit)
+ execute()
+ wasSuccessful() : boolean
+ addExecutionOperation(OperationCommand)
+ addAssertionOperation(TestAssertion)
testAssertionOperations
PersistentAssertionStrategy
+ getAssertionMethod() : Method
+ setAssertionMethod(Method)
- getPersistentAssertionMethod() : PersistentMethod
- setPersistentAssertionMethod(PersistentMethod)
persistentMethod
PersistentMethod
+ name: char[]
+ parameterClassNames: char[][]
+ declaringClassName: char[]
persistentOperation
PersistentOperationCommand
+ getTarget() : Object
+ setTarget(Object)
+ getOperation() : Method
+ setOperation(Method)
+ getParameters() : Object[]
+ setParameters(Object[])
+ getResult() : Object
+ setResult(Object)
+ getResultAccessorOperations() : Method[]
+ setResultAccessorOperations(Method[])
- getPersistentTarget() : PersistentObject
- setPersistentTarget(PersistentObject)
- getPersistentOperation() : PersistentMethod
- setPersistentOperation(PersistentMethod)
- getPersistentParameters() : PersistentObject[]
- setPersistentParameters(PersistentObject[])
- getPersistentResult() : PersistentObject
- setPersistentResult(PersistentObject)
- getPersistentResultAccessorOperations() : PersistentMethod[]
- setPersistentResultAccessorOperations(PersistentMethod[])
persistentResultAccessorOperations
target
result
parameters
assertionOperation
Object
expectedResult
actualResult
SingleTestAssertion
+ assertResult(DynamicTestCase)
+ getActualResult() : Object
+ setActualResult(Object)
+ getExpectedResult() : Object
+ setExpectedResult(Object)
TestAssertion
+ assertResult(DynamicTestCase)
GroupTestAssertion
+ assertResult(DynamicTestCase)
PersistentSingleTestAssertion
+ getActualResult() : Object
+ setActualResult(Object)
+ getExpectedResult() : Object
+ setExpectedResult(Object)
- getPersistentActualResult() : PersistentObject
- setPersistentActualResult(PersistentObject)
- getPersistentExpectedResult() : Object
- setPersistentExpectedResult(PersistentObject)
persistentTarget
persistentResult
persistentParameters
persistentActualResult
persistentExpectedResult
PersistentObject
+ value: char[]
+ className: char[]
+ setObject(Object)
+ getObject() : Object
PersistentDTO
+ addProperty(char[], char[])
+ getProperty(char[]) : char[]
properties
Property
+ name: char[]
+ value: char[]
+ getName() : char[]
+ getValue() : char[]
+ setName(char[])
+ setValue(char[])
serializationStrategy
SerializationStrategy
+ serializeValue(Object) : char[]
ServiceSerializationStrategy
+ serializeValue(Object) : char[]
assertionStrategy

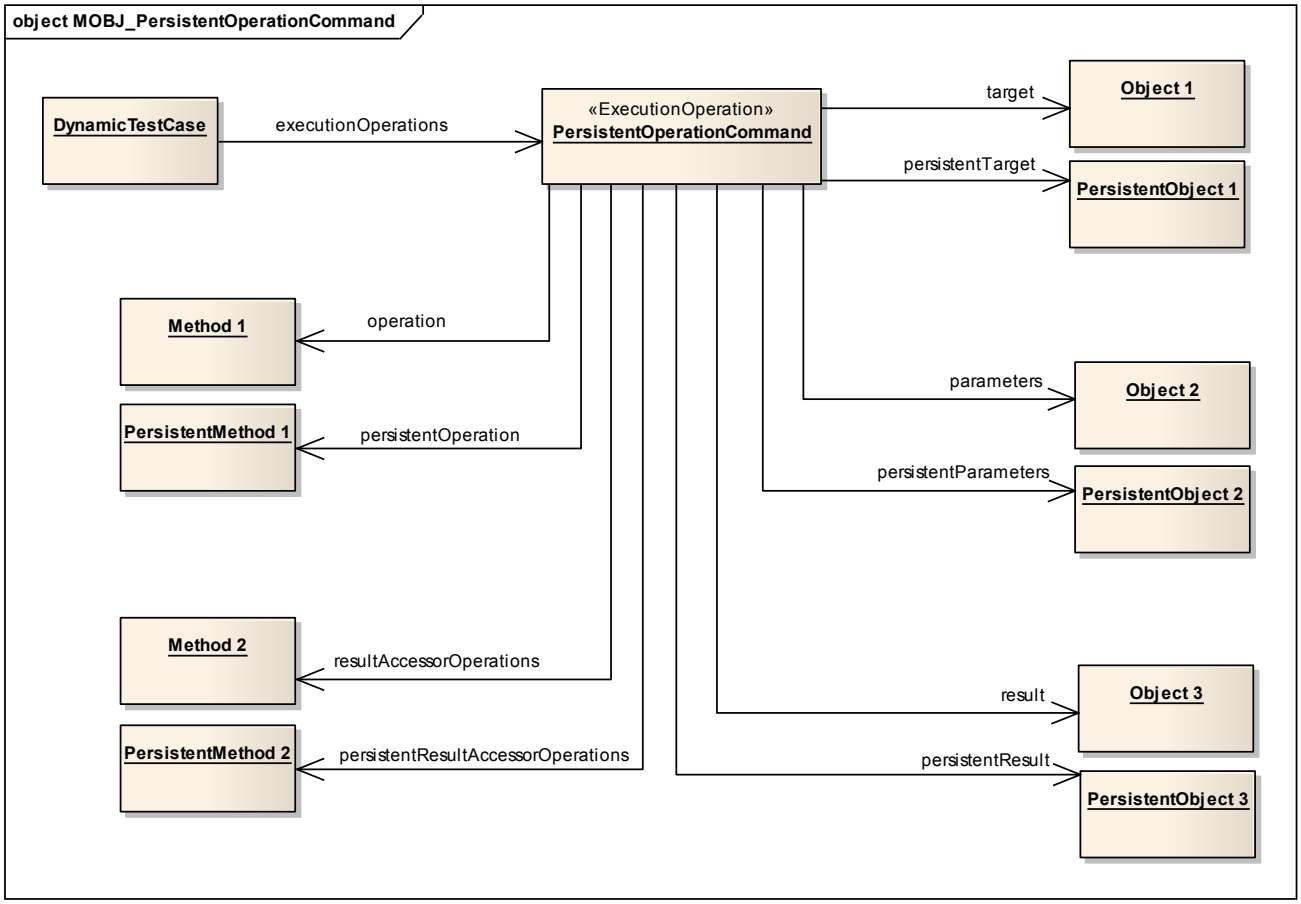


Figura 7.10 – Instanciación de un PersistentOperationCommand

En cuanto a estructura, cada una de estas subclases hereda todo de su superclase, y sólo agrega el conocimiento a PersistentObjects y PersistentMethods que constituyen la versión persistente de cada uno de los atributos Object y Method de su superclase. De esta manera, se obtiene una versión espejada y persistente de tales atributos. En cuanto a la persistencia en Hibernate, cada una de estas jerarquías se mapeará mediante la técnica de "varias clases a una tabla" [Hibernate], de forma que cada una de las tres tablas resultantes contendrá todos los atributos de las clases originales del modelo, salvo los de tipo Object y Method, que serán reemplazados por los definidos en sus subclases, de tipo PersistentObject y PersistentMethod.

En cuanto a la dinámica, en la versión persistente del modelo se utilizarán instancias de estas tres componentes (PersistentOperationCommand, PersistentSingleTestAssertion y PersistentAssertionStrategy) en lugar de instancias de las clases originales, pero gracias al principio de sustitución mencionado, el modelo tratará a las instancias de estas nuevas clases con el mismo protocolo definido en sus superclases y seguirá tipándolas con ellas, con lo cual el tratamiento de estas nuevas instancias es absolutamente transparente y nada invasivo. A continuación se describe en detalle cómo se consigue este dinamismo, es decir cómo se gestionan estas nuevas instancias y como se relacionan con las distintas componentes del modelo.

La solución se basa en la idea de "Interceptor". Un interceptor es un objeto que interfiere el flujo normal de una aplicación realizando algún tipo de acción lateral según su fin, y permitiendo luego la continuidad normal del flujo de ejecución. Es fundamental que esta intercepción sea transparente para la aplicación y no altere su flujo original de ejecución.

En el modelo, los interceptors son las tres nuevas subclases antes mencionadas, y los

puntos de intercepción que éstas definen son los getters y setters de los atributos de tipo Object y Method. Como se ha mencionado y tal como está definido en el modelo básico (ya que el mismo no se altera en absoluto), tales atributos están tipados con las clases Object y Method, pero gracias al principio de sustitución de Liskov las instancias que referencian corresponderán a las clases PersistentObject y PersistentMethod, respectivamente. De esta manera, el comportamiento de la mayoría de los mensajes que estas instancias reciban será heredado directamente de sus superclases. Sin embargo, al recibir un mensaje getter o setter tipado con Object o Method, los cuales sí implementan ya que los mismos funcionan como interceptors, ejecutan la lógica lateral de persistencia, para luego continuar con la ejecución normal del método heredada de su superclase.

Esta lógica lateral que ejecutan, consiste simplemente en ejecutar las operaciones de persistencia o regeneración del atributo, según se trate de un setter o un getter, respectivamente. A continuación se detallan los pasos concretos que ejecuta cualquiera de las sublclases (clases interceptors) al ejecutar un setter y un getter.

Ejecución de un setter (con un atributo de tipo Object):

1. El modelo envía un mensaje setter con un Object o como parámetro a una instancia de alguna de las subclases mencionadas.
2. La instancia, actuando como interceptor, ejecuta el comportamiento implementado en su clase (que es una de las subclases mencionadas). Este comportamiento no es ni más ni menos que la ejecución de la operación de persistencia para el parámetro o recibido. De esta manera, crea un PersistentObject que representa a o y setea este nuevo objeto en su variable de instancia que representa la versión persistente de tal atributo. Este nuevo PersistentObject ahora será automáticamente persistido por Hibernate.
3. De forma inmediatamente posterior a esta intercepción, la instancia deja correr el flujo normal de la aplicación ejecutando el método setter original de forma completa, heredado de su superclase. De esta forma, la instancia se setea el Object o, tal como lo determina el modelo básico.
4. Como resultado de lo anterior, la instancia cuenta con dos variables de instancia seteadas:
    a. Una tipada con Object, heredada de la clase original del modelo, y que ahora referencia a o.
    b. Una tipada con PersistentObject, definida en la subclase correspondiente del modelo extendido, y que ahora contiene la versión persistente de o.

Ejecución de un getter (que devuelve un atributo de tipo Object):

1. El modelo envía un mensaje getter a una instancia de alguna de las subclases mencionadas.
2. La instancia, actuando como interceptor, ejecuta el comportamiento implementado en su clase (que es una de las subclases mencionadas). Este comportamiento no es ni más ni menos que la ejecución de la operación de regeneración para su atributo. Esta regeneración toma los valores del PersistentObject que conoce (y que está persistido mediante Hibernate) y genera un o que setea a su atributo correspondiente tipado con Object.
3. De forma inmediatamente posterior a esta intercepción, la instancia deja correr el flujo normal de la aplicación ejecutando el método getter original de forma completa, heredado de su superclase. De esta forma, la instancia devuelve el objeto o que ha regenerado y seteado en el

paso anterior..

Si bien en los ejemplos anteriores se mostró la secuencia de pasos correspondiente a getters y setters para Object, el procedimiento de getter y setters para Method es exactamente igual, sólo que con instancias de PersistentMethod y operaciones de persistencia y regeneración correspondientes a métodos, que han sido descriptas en secciones anteriores.

```
public class PersistentSingleTestAssertion extends SingleTestAssertion {

    /** Versión persistente del expected result (Object) */
    PersistentObject persistentExpectedResult;

    [....]

    /**
     * Método interceptor.
     */
    public void setExpectedResult(Object result){

        // Intercepto el mensaje.
        PersistentObject persistentExpectedResult;
        persistentExpectedResult = result==null?null:new PersistentObject(result);
        this.setPersistentExpectedResult(persistentExpectedResult);

        // Continúa la ejecución normal del método.
        super.setExpectedResult(result);
    }


    /**
     * Método interceptor.
     */
    public Object getExpectedResult() throws Exception{

        // Intercepto el mensaje.
        if (super.getExpectedResult() == null & this.getPersistentExpectedResult() != null){
            // El atributo result de tipo Object es nulo, pero tiene información persistida.
            // Lazy-initialization: se carga el Object a partir de su información persistida.
            super.setExpectedResult(this.getPersistentExpectedResult().getObject());
        }

        // Continúa la ejecución normal del método.
        return super.getExpectedResult();
    }

    [....]
```

Figura 7.11 – Ejemplo de Métodos Interceptores

7.6. Creación transparente de componentes de testing

Hasta aquí se ha descripto cómo los Objects y Methods pueden persistirse a través de PersistentObjects y PersistentMethods, y cómo las componentes del modelo básico con atributos tipados con aquellas clases pueden persistirse transparentemente a través de las subclases interceptors. En esta sección se describirá un último recaudo que debe considerarse para lograr la persistencia del modelo completo de forma totalmente transparente y evitando la invasión del mismo.

Tal como se ha descripto hasta aquí, si el modelo básico se instancia sin persistencia (o mediante una técnica de persistencia absolutamente transparente, a diferencia de Hibernate)

todas las clases del modelo básico pueden instanciarse de forma directa. Por otro lado, si el modelo se persiste con Hibernate mediante la técnica descripta en las secciones anteriores, las instancias de algunas de las clases del modelo básico (OperationCommand, SingleTestAssertion y AssertionStrategy) deben crearse a partir de sus subclases (versión persistente) y no de ellas mismas (en este contexto, ellas sirven a los efectos de brindar estructura y comportamiento, pero no operaciones de creación).

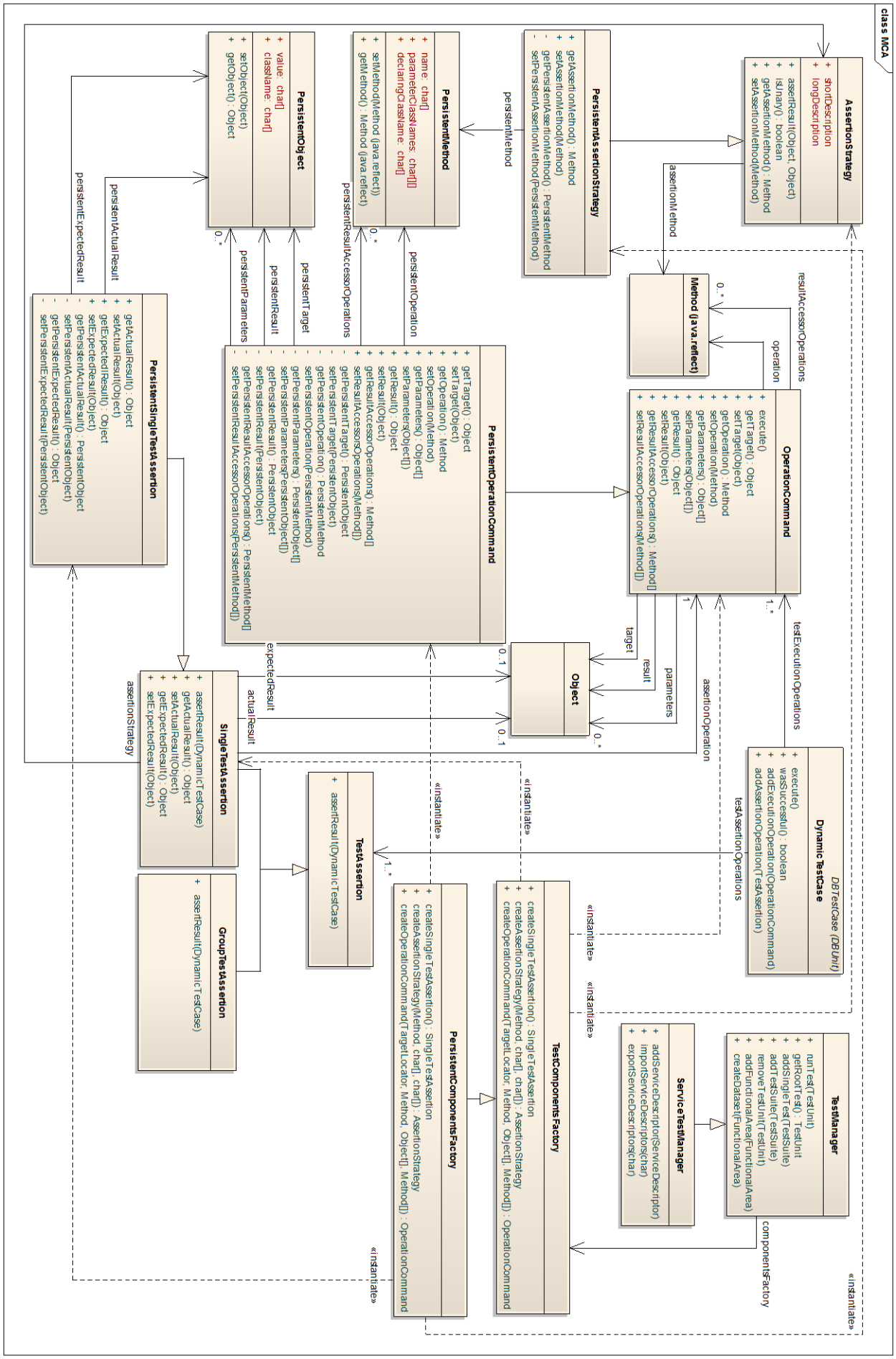
Entonces en este punto surge la siguiente cuestión: ¿Cómo se especifica la forma de creación de estas componentes en el modelo básico, de forma tal de no especificar en su diseño y código clases de extensión que no hacen a su esencia?

En algún punto, por ejemplo, el modelo deberá decidir si crea un OperationCommand o un PersistentOperacionCommand, según el modelo esté o no persistido. Y siguiendo los lineamientos de transparencia y no intrusión establecidos en el presente trabajo, el modelo básico no podría nunca referenciar a clases que no hacen a su esencia y pertenecen a contextos más particulares; por ejemplo, no debe referenciar a la clase PersistentOperationCommand.

La solución de diseño seleccionada para resolver esta situación es la aplicación del patrón de diseño Factory. Una clase Factory toma la responsabilidad de crear un conjunto de componentes, para lo cual provee un protocolo determinado. A su vez, pueden tenerse varias clases Factory que compartan el mismo protocolo de creación de objetos, pero que lo implementen de manera diferente.

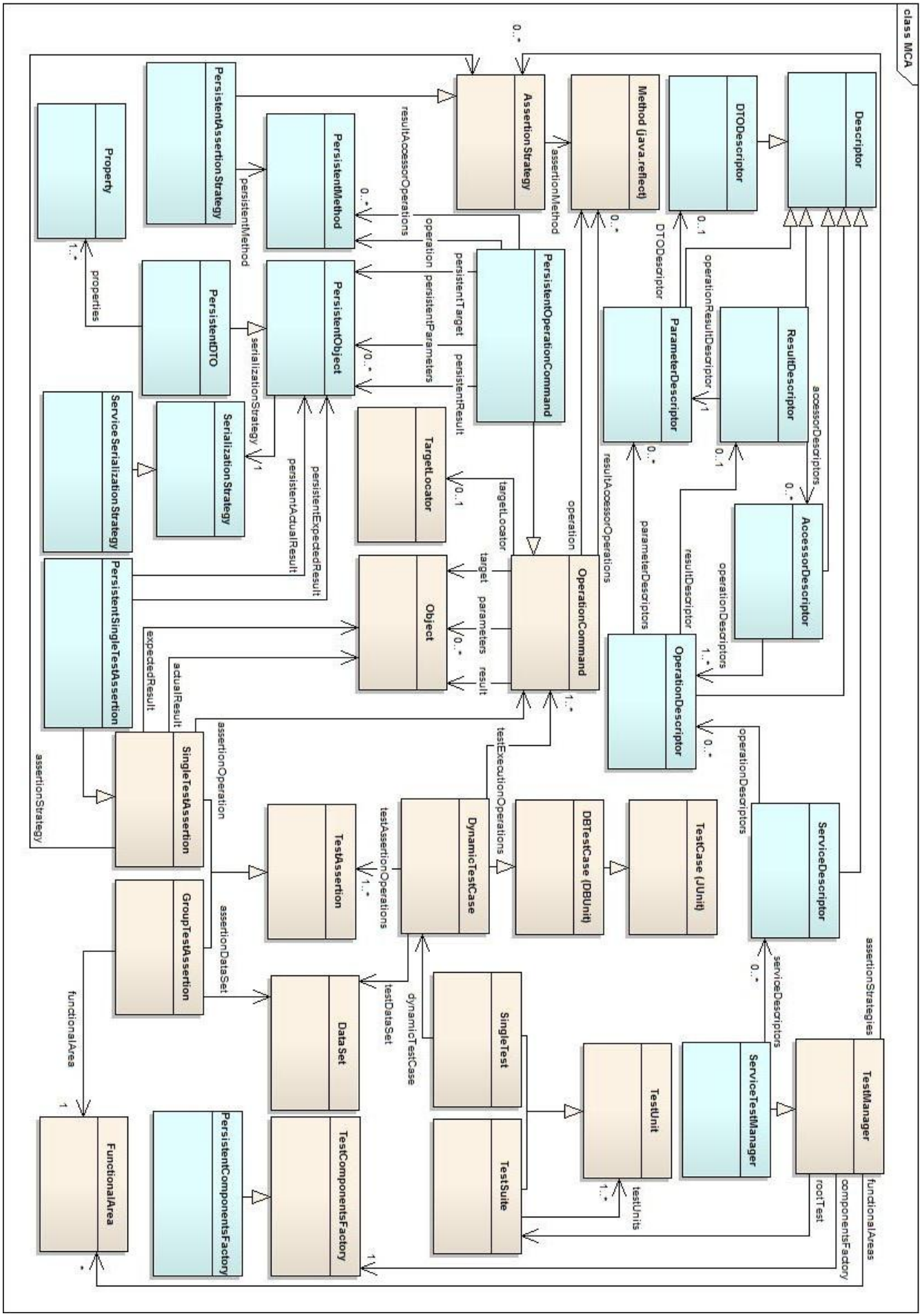
En el contexto de este trabajo, pueden aplicarse este patrón para la creación de algunas de las instancias componentes del modelo básico, justamente las previamente descriptas (OperationCommand, SingleTestAssertion y AssertionStrategy) cuya creación varía según el modelo se persista o no. En este sentido, pueden crearse dos clases Factory, ambas con idéntico protocolo de creación de componentes, y ambas tipando con las superclases mencionadas (las del modelo básico) a las instancias creadas y devueltas. Sin embargo, la diferencia entre ambas radica en su implementación ya que una de ellas creará instancias de las clases del modelo básico, mientras que la otra creará instancias de las subclases que proveen funcionalidad de persistencia tal como se describió en las secciones anteriores.

El modelo básico podrá ser configurado mediante inyección de dependencias con uno u otro Factory y se comunicará de forma homogénea con el mismo, sea cual fuere, cuando necesite crear alguna componente. Cada Factory devolverá una u otra instancia (subclase o clase original) según el caso, pero siempre tipada con la clase del modelo básico original, obteniendo así la transparencia establecida como lineamiento de este trabajo. En el modelo, estas dos Factories se implementan mediante las clases TestComponentsFactory y PersistentComponentsFactory.


class MCA
AssertionStrategy
+ shortDescription
+ longDescription
+ assertResult(Object, Object)
+ isUnary() : boolean
+ getAssertionMethod() : Method
+ setAssertionMethod(Method)
PersistentAssertionStrategy
+ getAssertionMethod() : Method
+ setAssertionMethod(Method)
- getPersistentAssertionMethod() : PersistentMethod
- setPersistentAssertionMethod(PersistentMethod)
persistentMethod
assertionMethod
Method (java.reflect)
resultAccessorOperations
operation
OperationCommand
+ execute()
+ getTarget() : Object
+ setTarget(Object)
+ getOperation() : Method
+ setOperation(Method)
+ getParameters() : Object[]
+ setParameters(Object[])
+ getResult() : Object
+ setResult(Object)
+ getResultAccessorOperations() : Method[]
+ setResultAccessorOperations(Method[])
testExecutionOperations
DBTestCase (DBUnit)
Dynamic TestCase
+ execute()
+ wasSuccessful() : boolean
+ addExecutionOperation(OperationCommand)
+ addAssertionOperation(TestAssertion)
testAssertionOperations
PersistentMethod
+ name: char[]
+ parameterClassNames: char[][]
+ declaringClassName: char[]
+ setMethod(Method (java.reflect))
+ getMethod() : Method (java.reflect)
persistentResultAccessorOperations
persistentOperation
PersistentObject
+ value: char[]
+ className: char[]
+ setObject(Object)
+ getObject() : Object
persistentActualResult
persistentExpectedResult
persistentParameters
persistentResult
persistentTarget
PersistentOperationCommand
+ getTarget() : Object
+ setTarget(Object)
+ getOperation() : Method
+ setOperation(Method)
+ getParameters() : Object[]
+ setParameters(Object[])
+ getResult() : Object
+ setResult(Object)
+ getResultAccessorOperations() : Method[]
+ setResultAccessorOperations(Method[])
- getPersistentTarget() : PersistentObject
- setPersistentTarget(PersistentObject)
- getPersistentOperation() : PersistentMethod
- setPersistentOperation(PersistentMethod)
- getPersistentParameters() : PersistentObject[]
- setPersistentParameters(PersistentObject[])
- getPersistentResult() : PersistentObject
- setPersistentResult(PersistentObject)
- getPersistentResultAccessorOperations() : PersistentMethod[]
- setPersistentResultAccessorOperations(PersistentMethod[])
Object
target
result
parameters
assertionOperation
expectedResult
actualResult
assertionStrategy
PersistentSingleTestAssertion
+ getActualResult() : Object
+ setActualResult(Object)
+ getExpectedResult() : Object
+ setExpectedResult(Object)
- getPersistentActualResult() : PersistentObject
- setPersistentActualResult(PersistentObject)
- getPersistentExpectedResult() : Object
- setPersistentExpectedResult(PersistentObject)
SingleTestAssertion
+ assertResult(DynamicTestCase)
+ getActualResult() : Object
+ setActualResult(Object)
+ getExpectedResult() : Object
+ setExpectedResult(Object)
GroupTestAssertion
+ assertResult(DynamicTestCase)
TestAssertion
+ assertResult(DynamicTestCase)
«instantiate»
PersistentComponentsFactory
+ createSingleTestAssertion() : SingleTestAssertion
+ createAssertionStrategy(Method, char[], char[]) : AssertionStrategy
+ createOperationCommand(TargetLocator, Method, Object[], Method[]) : OperationCommand
TestComponentsFactory
+ createSingleTestAssertion() : SingleTestAssertion
+ createAssertionStrategy(Method, char[], char[]) : AssertionStrategy
+ createOperationCommand(TargetLocator, Method, Object[], Method[]) : OperationCommand
componentsFactory
TestManager
+ runTest(TestUnit)
+ getRootTest() : TestUnit
+ addSingleTest(TestSuite)
+ addTestSuite(TestSuite)
+ removeTestUnit(TestUnit)
+ addFunctionalArea(FunctionalArea)
+ createDataset(FunctionalArea)
ServiceTestManager
+ addServiceDescriptor(ServiceDescriptor)
+ importServiceDescriptors(char)
+ exportServiceDescriptors(char)

7.7. Modelo Extendido Completo

Con todo lo descripto hasta aquí, el modelo básico queda persistido de manera absolutamente transparente y sin ser invadido en ninguno de sus puntos. Este modelo extendido completo se muestra en la siguiente figura. Por razones de espacio, se omite la información de atributos y mensajes de cada clase, los cuales pueden ser consultados en los diagramas anteriores que se han mostrado durante el desarrollo de la descripción del modelo.


class MCA
Descriptor
DTODescriptor
Method (java.reflect)
AssertionStrategy
PersistentAssertionStrategy
Property
PersistentMethod
PersistentDTO
PersistentObject
PersistentOperationCommand
ParameterDescriptor
ResultDescriptor
AccessorDescriptor
ServiceSerializationStrategy
SerializationStrategy
TargetLocator
Object
OperationCommand
OperationDescriptor
PersistentSingleTestAssertion
SingleTestAssertion
GroupTestAssertion
TestAssertion
DynamicTestCase
DBTestCase (DBUnit)
TestCase (JUnit)
ServiceDescriptor
DataSet
SingleTest
TestSuite
TestUnit
ServiceTestManager
TestManager
FunctionalArea
PersistentComponentsFactory
TestComponentsFactory
resultAccessorOperations
operation
persistentTarget
persistentParameters
persistentResult
persistentMethod
properties
serializationStrategy
persistentActualResult
persistentExpectedResult
targetLocator
target
parameters
result
DTODescriptor
operationResultDescriptor
accessorDescriptors
operationDescriptors
resultDescriptor
parameterDescriptors
testExecutionOperations
assertionOperation
assertionStrategy
expectedResult
actualResult
testAssertionOperations
testDataSet
dynamicTestCase
assertionDataSet
functionalArea
serviceDescriptors
assertionStrategies
functionalAreas
componentsFactory
rootTest
testUnits

8. IMPLEMENTACIÓN Y GUIS DE LAS APLICACIONES COMPONENTES

En este capítulo se describen los aspectos técnicos de arquitectura, implementación e interfaces de usuario de las aplicaciones desarrolladas para este trabajo: Aplicación de Testing y Aplicación Objetivo.

Cabe destacar nuevamente que la Aplicación de Testing es única y se utiliza para testear cualquier Aplicación Objetivo de tipo SOA cuya arquitectura cumpla con las generalidades descriptas en capítulos anteriores. En cuanto a la Aplicación Objetivo que se toma como ejemplo y se presenta en este trabajo, se trata, en particular, de una implementación reducida del sistema SCUT previamente descripto, actualmente utilizado en la TGP.

8.1. Aplicación Objetivo (SCUT)

En esta sección se describe la aplicación SCUT implementada para este trabajo y entregada como parte del mismo como ejemplo de aplicación objetivo.

Esta aplicación da soporte al dominio financiero previamente descripto y se enfoca sobre el recorte consistente en gestionar a los Organismos y Cuentas Escriturales de la administración provincial, permitiendo realizar operaciones sobre las mismas.

En primer lugar se describe su arquitectura en cuanto a la tecnología utilizada, y a continuación la utilización de la aplicación a través de su interfaz gráfica.

8.1.1. Arquitectura de la aplicación

La arquitectura de esta aplicación es como la ya descripta en general para las aplicaciones objetivo de tipo SOA. A continuación se describen brevemente sus capas principales, aludiendo a las tecnologías concretas que se han utilizado en su construcción.

8.1.1.1. Capa de Presentación

La Capa de Presentación está constituida por una interfaz gráfica, una aplicación cliente tipo desktop construida en Java mediante Swing. Cabe recordar que la naturaleza SOA de la arquitectura objetivo permite independizar la construcción de la interfaz del modelo subyacente, ya que ésta se comunica con aquél mediante la capa de servicios. De esta manera podría crearse una interfaz mediante otra tecnología, por ejemplo web, sin alterar en lo más mínimo el modelo ni la capa de servicios. La elección de tecnología Java/Swing se ha realizado sólo a los fines de facilitar la construcción y el testeo, considerando a su vez que la interfaz no conforma el punto central de este trabajo.

8.1.1.2. Capa de Servicios

La Capa de Servicios está constituida por un conjunto de clases Java cuyas instancias implementan los servicios exportados. Estas clases implementan una interfaz (procotolo de mensajes), de forma tal de separar la definición de la implementación de estos servicios.

Estos servicios son invocados por cualquier parte habilitada a interactuar con el modelo, por ejemplo, la interfaz descripta en la sección anterior. Esta capa, para dar respuesta a estos servicios, interactúa con el modelo subyacente y, a su vez, para no exponerlo y dar soluciones a temas transaccionales, no exporta componentes del modelo sino DTOs, que se encargan de transportar información entre la capa de servicios y quienes la utilizan.

En concreto, se proveen dos tipos de servicios, los relacionados a Organismos y los referentes a Cuentas Escriturales. A continuación se muestra un extracto de la definición y

exportación de la interfaz para cuentas escriturales:

```
/**
 * Interfaz de servicios provistos para Cuentas Escriturales.
 */
public interface ICuentaEscrituralServicio {


    /**
     * Deposita el monto recibido en la cuenta escritural especificada.
     */
    public void depositar(CuentaEscrituralDTO cuentaEscrituralDTO, float monto);


    /**
     * Extrae el monto recibido de la cuenta escritural especificada.
     */
    public void extraer(CuentaEscrituralDTO cuentaEscrituralDTO, float monto);


    /**
     * Transfiere el monto recibido entre las cuentas escriturales especificadas.
     */
    public void transferir(CuentaEscrituralDTO cuentaEscrituralOrigenDTO,
                           CuentaEscrituralDTO cuentaEscrituralDestinoDTO,
                           float monto);

    [....]
```

Figura 8.1 – Interfaz de Servicios de Cuentas Escriturales

Las clases que implementan estos servicios son OrganismoServiceImpl y CuentaEscrituralServiceImpl.

Quienes utilicen estos servicios (por ejemplo la aplicación cliente) deberán poder obtener una referencia a estos objetos para poder ejecutar las operaciones que definen. Para efectuar la localización de estos servicios se provee un localizador de servicios, implementado mediante la clase ServiceLocator.

Esta clase se ha diseñado implementando el patrón de diseño Singleton, de forma tal que existe una instancia de la misma y tal instancia se obtiene pidiéndosela a la clase. De esta forma se facilita y permite el acceso a los servicios para quienes necesiten utilizarlos.

8.1.1.3. Capa del Modelo

La Capa del Modelo define las clases básicas que permiten el funcionamiento del recorte del dominio financiero presentado. Este modelo permite básicamente la gestión de Cuentas Escriturales y Organismos provinciales.

Las Cuentas Escriturales poseen como titular a un Organismo provincial, proveen las operaciones de depósito, extracción, transferencia, consulta de saldo y movimientos que tales transacciones generan. A su vez, cada movimiento referencia a su tipo, que identifica la naturaleza del mismo.

La clase SCUT es la responsable de mantener y gestionar a las anteriores, y se constituye como punto de entrada al modelo, aplicándose así el patrón de diseño Facade. Las implementaciones de los servicios de la capa anterior accederán al modelo a través de la única instancia de esta clase (patrón Singleton).

A continuación se muestra el diagrama de clases de la capa del modelo.

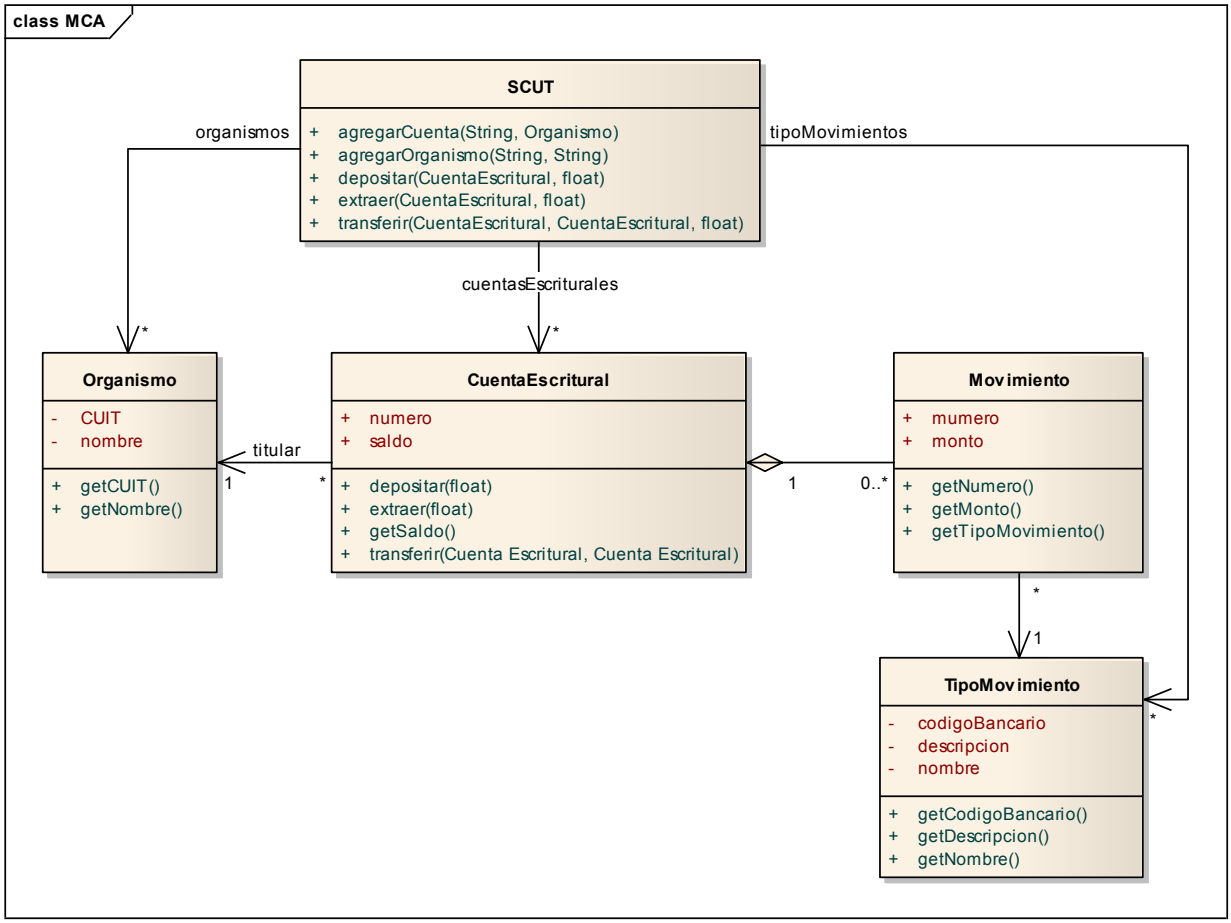


Figura 8.2 – Modelo de clases del SCUT (Aplicación Objetivo)

8.1.1.4. Capa de Persistencia

La capa de persistencia se implementa mediante una base de datos relacional, para lo cual se ha seleccionado MySQL por disponer de una versión gratuita para su uso no comercial. En esta base de datos radican todas las tablas que dan soporte persistente a las clases del modelo.

Se utiliza también un mapeador O/R para gestionar la diferencia entre estos dos paradigmas (relacional y orientado a objetos) tanto en estructura como en queries, para lo cual se ha seleccionado Hibernate por ser uno de los más utilizados en el mercado, y particularmente en el contexto de la aplicación de producción real en la TGP.

8.1.1.5. Arquitectura completa

A continuación se muestra un esquema de la arquitectura completa, considerando las tecnologías concretas utilizadas.

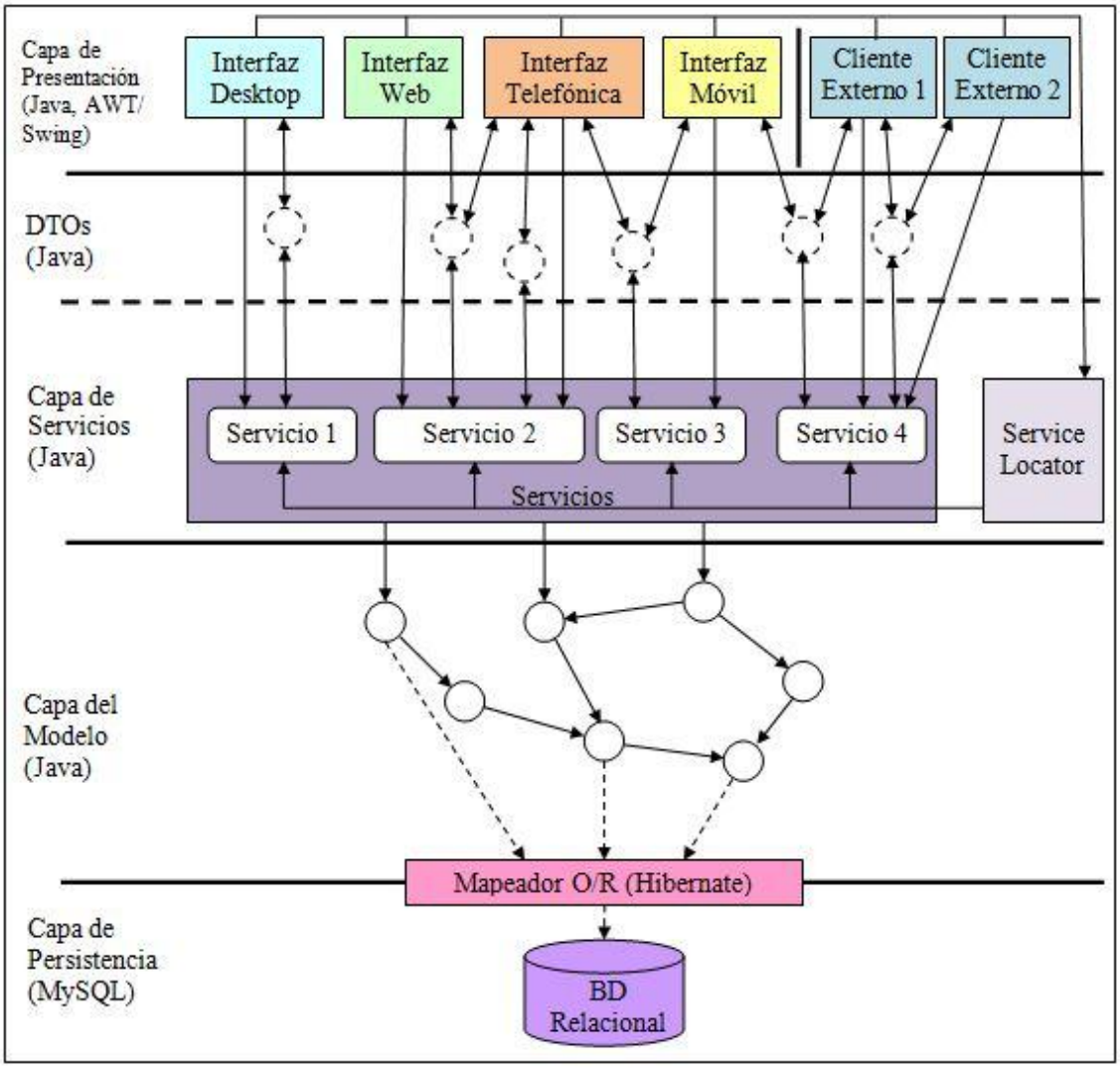


Figura 8.3 – Arquitectura completa de la Aplicación Objetivo

Además de las tecnologías anteriormente descriptas se utilizan otras como XML y JAXB para la gestión de archivos de configuración e importación/importación desde y hacia la capa del modelo y servicios. Asimismo se utiliza Spring [Spring] para la integración de las distintas tecnologías, haciendo especial uso de la inyección de dependencias para una gestión transparente de los colaboradores de varios objetos.

8.1.2. Uso de la aplicación

En esta sección se explicará el uso de la aplicación a través de la interfaz gráfica, desde la perspectiva que un usuario final tiene de la misma para su uso cotidiano.

Tal como se explicó anteriormente, esta interfaz (Capa de Presentación) invoca las operaciones provistas por la Capa de Servicios e intercambia información con ésta a través de DTOs, por lo que se mantiene aislada del modelo.

8.1.2.1. Ventana Principal

Al iniciar la aplicación se abre la pantalla principal, que se muestra a continuación. Esta

pantalla, a través de sus menúes, permite el acceso a las funcionalidades de gestión de Organismos y Cuentas Escriturales.

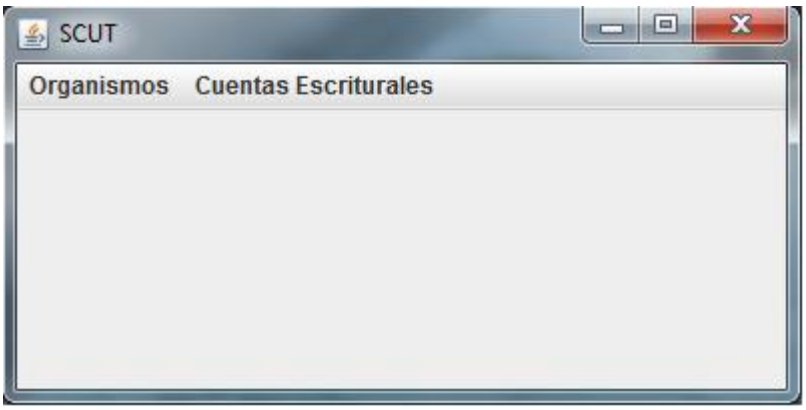


Figura 8.4 – Pantalla Principal

8.1.2.2. Gestión de Organismos

Esta gestión permite en primer lugar dar de alta un organismo a través de la opción de menú Organismos > Nuevo de la pantalla principal. Para el alta de un Organismo debe especificarse su nombre y su CUIT, mediante la siguiente ventana:

Figura 8.5 – Alta de Organismo

Los organismos dados de alta en el sistema pueden consultarse a través de una ventana de listado que se despliega mediante la opción Organismos > Listar de la ventana principal.

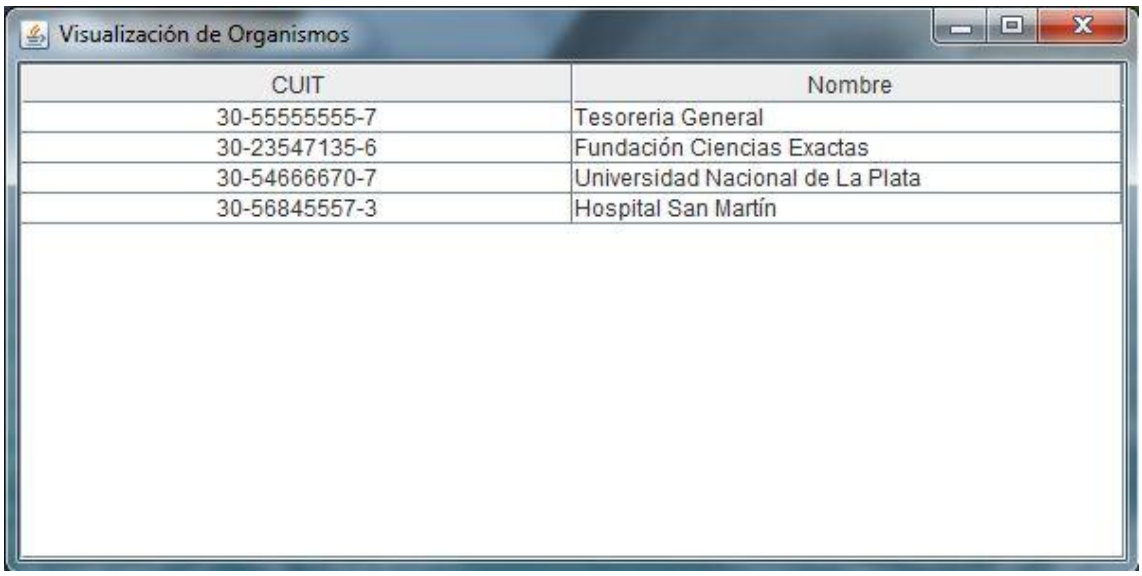


Figura 8.6 – Listado de Organismos

Tal como puede observarse, mediante el botón derecho se abre un menú contextual que permite la eliminación del organismo seleccionado. Tal organismo será eliminado sólo si no es titular de ninguna cuenta escritural, en cuyo caso debe primero eliminarse ésta.

8.1.2.3. Gestión de Cuentas Escriturales

El alta de una nueva escritural se realiza desde la opción Cuentas Escriturales > Nueva de la ventana principal, la que despliega la siguiente ventana.

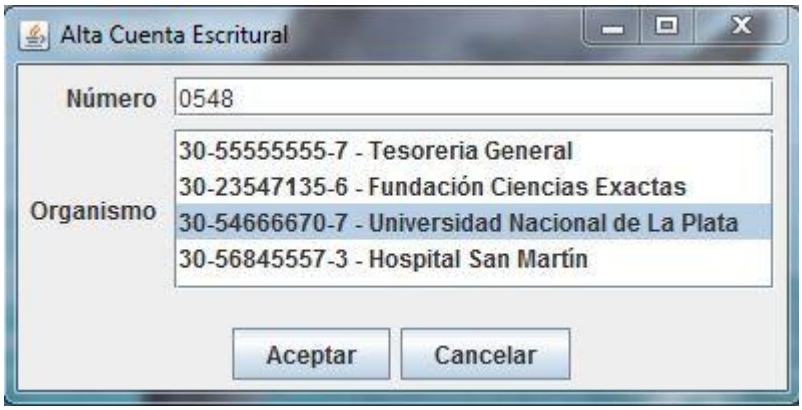


Figura 8.7 – Alta de Cuenta Escritural

Esta ventana permite el ingreso del número de cuenta que el usuario desea asignarle como así también el organismo titular de la misma, para lo cual se ofrece al usuario la lista de organismos cargados en el sistema para que seleccione uno de ellos. Al efectivizarse el alta de la cuenta se la inicializa con saldo cero y una lista vacía de movimientos.

Sobre las cuentas cargadas en el sistema pueden realizarse un conjunto de operaciones, tales como depósito, extracción y transferencia. Para esto debe accederse a la opción Cuentas

Escriturales > Operaciones de la ventana principal, lo cual despliega la siguiente ventana:

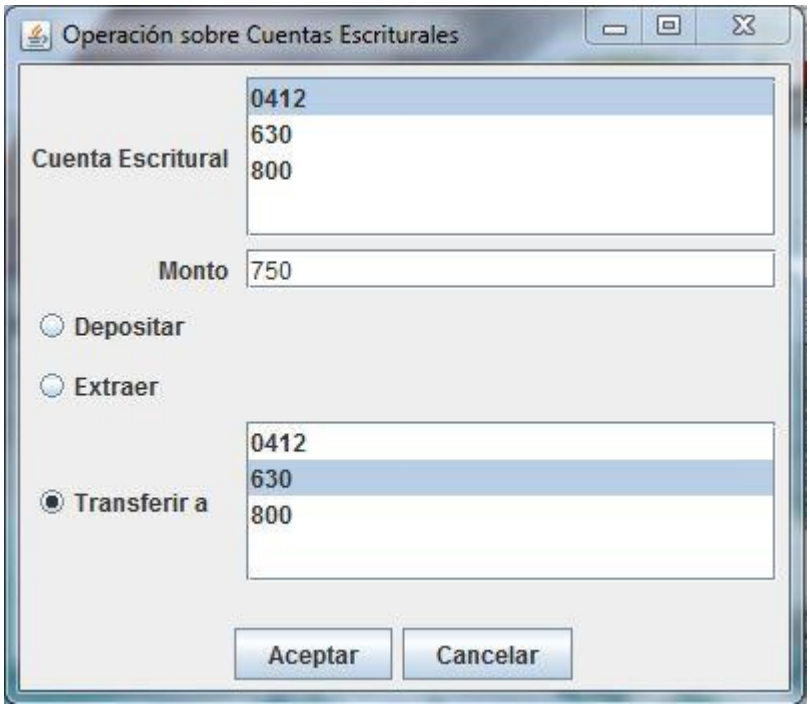


Figura 8.8 – Depósito, Extracción y Transferencia entre Cuentas Escriturales

En esta ventana se ofrece al usuario la/s cuenta/s escritural/es involucrada/s, el operación que desea realizar y el monto de tal operación. Por razones de simplicidad, se provee una única ventana para los tres tipos de operaciones que pueden realizarse sobre las cuentas. Al aceptar la operación, se agrega un nuevo movimiento a la/s cuenta/s correspondiente/s y se actualiza su salgo.

Como regla de negocio se establece que las operaciones de extracción y transferencias no tienen ningún efecto cuando el monto a extraer/transferir supera el saldo disponible de la cuenta.

Por último, a través de la opción Cuentas Escriturales > Listar de la ventana principal se despliega una ventana que permite ver todas las cuentas escriturales cargadas en el sistema. Para cada una se muestran sus datos básicos (número, titular, saldo) y, al seleccionar una de ellas, en la parte inferior de la pantalla se mostrarán los movimientos de la misma que, al haberse ejecutado en secuencia, conforman el saldo final de la cuenta.

Figura 8.9 – Listado de Cuentas Escriturales y sus movimientos

Como puede observarse, el botón derecho del mouse despliega un menú contextual que permite la eliminación de la cuenta escritural. Como regla de negocio, sólo se permite la eliminación de una cuenta escritural sólo si ésta no posee movimientos, ya que, en caso contrario, desde sus movimientos podría tener referencias cruzadas con otras cuentas escriturales a las o desde las cuales se haya transferido. Esta regla de negocio busca evitar referencias a cuentas inexistentes.

8.2. Aplicación de Testing

En esta sección se describe la aplicación de testing cuya implementación forma parte de este trabajo y que, como prueba de concepto, implementa una gran parte del modelo de testing descripto previamente. Esta aplicación cumple con los lineamientos de diseño ya descriptos en cuanto a flexibilidad y generalidad, ya que puede adaptarse a distintas aplicaciones objetivo de tipo SOA como las descriptas previamente.

Como se explicó anteriormente, esta aplicación de testing testea a la aplicación objetivo comunicándose con la misma a través de la capa de servicios provista por esta última.

En primer lugar se describe su arquitectura en cuanto a las tecnologías utilizadas, y a continuación la utilización de la aplicación a través de su interfaz gráfica.

8.2.1. Arquitectura de la aplicación

Si bien hay notables diferencias en cuanto a la generalidad lograda en las capas del modelo de esta aplicación con respecto a las de la aplicación objetivo descripta en la sección anterior, en el resto de las capas las arquitecturas de ambas son muy similares ya que las razones que justifican tales elecciones son las mismas que ya se han descripto. En esta línea, la aplicación de testing está compuesta por las siguientes capas:

- Capa de Presentación: implementada mediante una aplicación cliente en Java/Swing.
- Capa de Servicios: realizada en Java mediante objetos servicio, los cuales implementan una interfaz Java que define el protocolo de operaciones que la aplicación puede ejecutar. La transferencia de información entre esta capa y los usuarios de la misma se realiza mediante DTOs.
- Capa del Modelo: modelo de clases realizado en Java que implementa el Modelo Básico ya descripto en este trabajo. Posee un alto grado de generalidad y flexibilidad para adaptarse a distintas aplicaciones objetivo.
- Capa de Persistencia: implementada mediante una base de datos MySQL y Hibernate como mapeador O/R.

Asimismo se utiliza Swing como framework de aplicaciones para integrar las distintas tecnologías y, además, para implementar inyección de dependencias que es justamente una de las técnicas mediante las cuales se puede configurar a la aplicación de testing de forma transparente para adaptarla a distintas aplicaciones objetivo.

8.2.2. Uso de la aplicación

En esta sección se explicará el uso de la aplicación a través de la interfaz gráfica, desde la perspectiva que un usuario final tiene de la misma para su uso cotidiano, el cual consiste en la generación y ejecución de casos de tests de forma automática y amigable.

8.2.2.1. Ventana Principal

Al iniciar la aplicación se abre la pantalla principal que se muestra a continuación. En esta pantalla puede observarse un menú y un área de definición de tests; ambas se describen en esta sección.

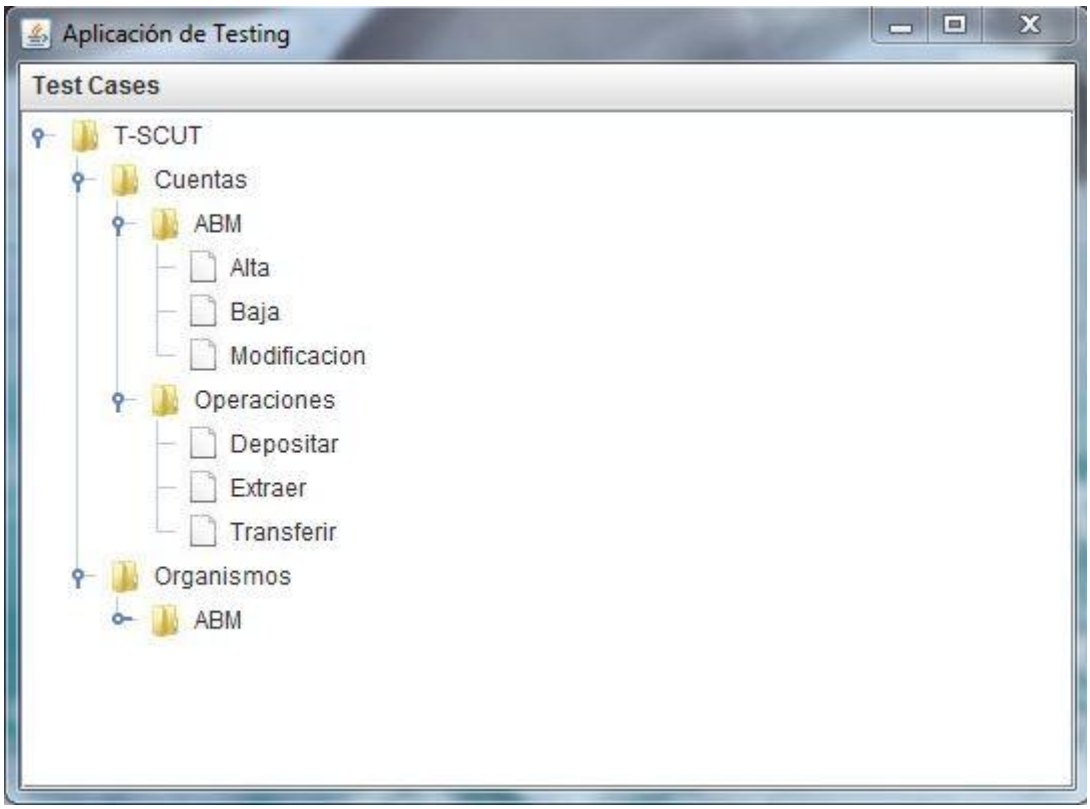


Figura 8.10 – Pantalla Principal

8.2.2.2. Barra de Menú

En el menú principal se encuentran un conjunto de opciones que permiten realizar una serie de importaciones y exportaciones. Las mismas se describen a continuación.

La opción Import Service Descriptors despliega una ventana que permite seleccionar un archivo .xml del file system que contiene los descriptores de servicios provistos por la aplicación objetivo y que se incorporarán a la aplicación de testing para ser utilizados posteriormente en la definición de tests. Este es uno de los puntos en que puede apreciarse el dinamismo y flexibilidad de adaptación de la aplicación de testing. Esta importación se realiza mediante la siguiente ventana:

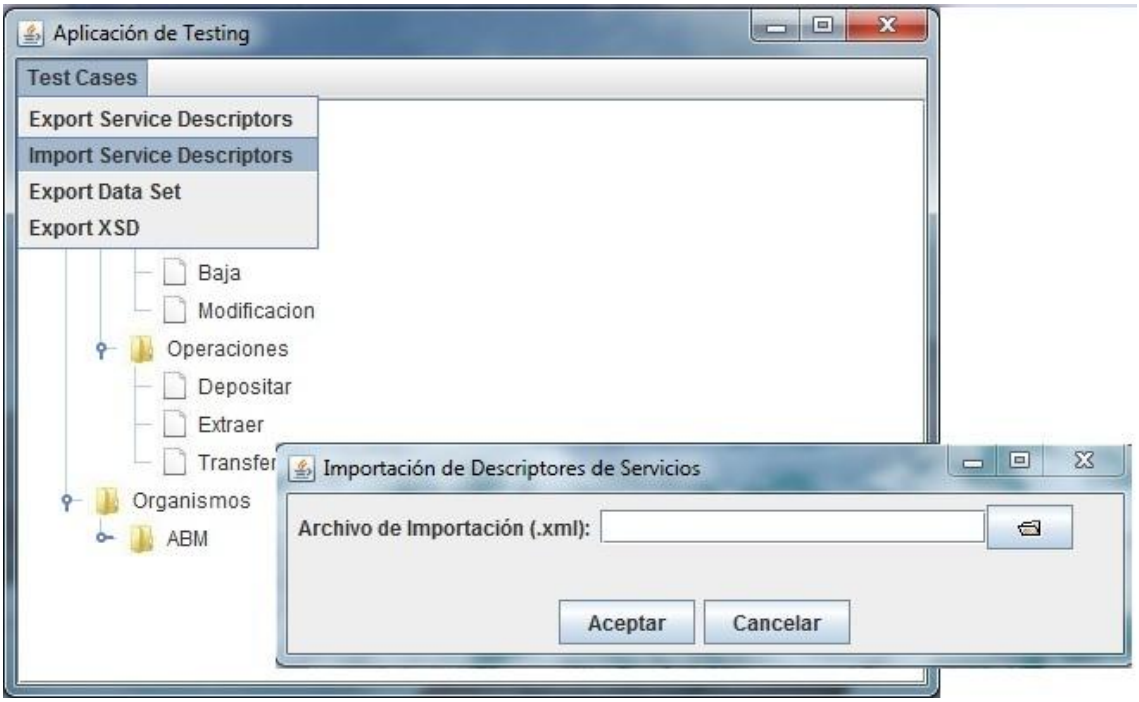


Figura 8.11 – Importación de Descriptores de Servicios

La opción Export Service Descriptors realiza la operación inversa: a partir del modelo de Descriptores de Servicios de la aplicación se genera un archivo .xml que representa su estructura e información. Esto es útil cuando se desea extender o modificar algún descriptor para luego ser importado nuevamente. Esta exportación se lleva a cabo mediante la siguiente ventana:

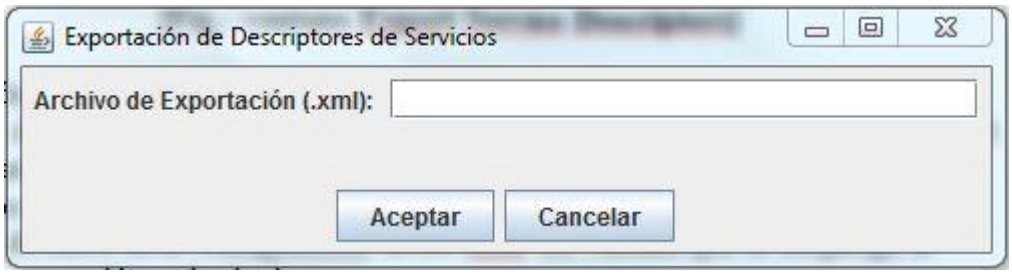


Figura 8.12 - Exportación de Descriptores de Servicios

La opción Export Dataset genera un juego de datos exportando a un archivo .xml la información de la base de datos de la aplicación objetivo. El usuario puede especificar el nombre del archivo .xml al cual se exportará la información y el área funcional de la base de datos que se desea exportar. Estos archivos son los que el usuario podrá utilizar como juego de datos en la configuración de los tests. La ventana que se despliega al seleccionar esta opción es la siguiente:

Figura 8.13 – Exportación de un escenario de datos

Por último, la opción Export XSD realiza la exportación de la definición de la estructura del dataset. Esta es una opción avanzada que carece de sentido para un usuario final pero puede ser útil para un usuario configurador con mayor conocimiento técnico, por lo que debe ser invisible para un usuario final.

8.2.2.3. Panel de definición de tests

En esta área puede visualizarse una estructura de árbol que permite ver fácilmente el anidamiento de la configuración de Tests y TestSuites definidos en la aplicación como así también el estado y resultado de ejecución de los mismos. Asimismo se pueden realizar operaciones mediantes menúes contextuales que se despliegan al cliquear el botón derecho del mouse sobre el elemento del árbol sobre el cual se desea operar.

Cada Test Suite está representado mediante una carpeta, cliqueando sobre la cual puede desplegarse o colapsarse su contenido, el cual está compuesto a su vez por otros Test Suites o Tests simples, representados estos últimos mediante una hoja de papel.

Cada uno de estos elementos puede estar en uno de tres estados de acuerdo a su estado y resultado de ejecución, los cuales se diferencian mediante distintos íconos gráficos, a saber:

- Check verde: significa que el elemento ha sido ejecutado y ha pasado el test.
- Signo rojo: significa que el elemento ha sido ejecutado y no ha pasado el test. Esto puede haber ocurrido tanto porque falló alguna de sus operaciones de aserción como porque ha ocurrido un error de cualquier índole.
- Sin marca: si el elemento no posee ninguna de las marcas anteriores significa que el mismo no ha sido ejecutado.

Estas marcas tienen relación jerárquica y recursiva entre sí. En este sentido, para que un nodo padre figure como “exitoso” (check verde), todos sus hijos deben figurar en el mismo estado. Por otro lado, habiéndose ejecutado todos sus nodos hijos, basta con que uno de ellos no haya pasado el test para que también su padre se considere fallado.

Por otro lado, se considera que un nodo padre está ejecutado cuando todos sus nodos hijos han sido ejecutados, independientemente del resultado; es por esto que basta con que un solo hijo esté “no ejecutado” (sin marca) para que determine el mismo estado en el nodo padre.

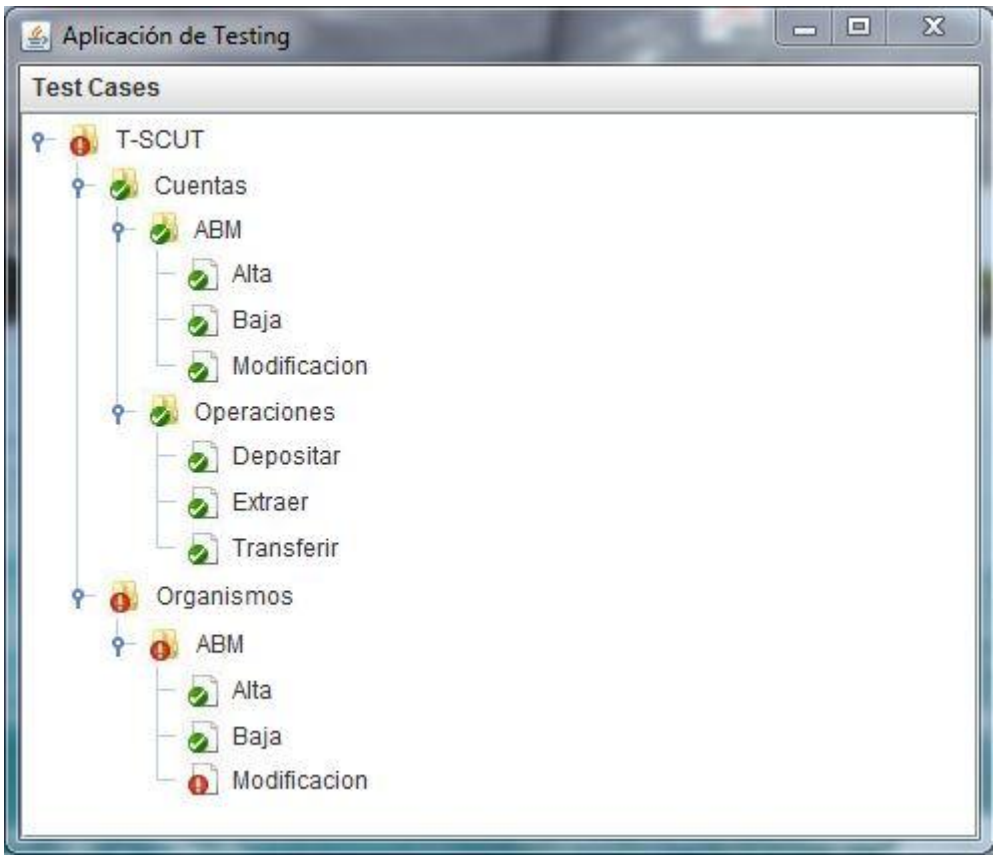


Figura 8.14 – Estados de ejecución de tests y test suites

En las próximas secciones se describen el conjunto de operaciones que pueden ser ejecutadas sobre los elementos del árbol a través de los menúes contextuales.

8.2.2.4. Creación de Tests y Test Suites

Para crear un Test o un Test Suite se debe presionar con el botón derecho del mouse sobre el Test Suite en el que se desea incluirlo (nodo padre); cabe destacar que la aplicación siempre estará inicializada con al menos con un Test Suite vacío como nodo raíz.

El menú contextual que se despliega ofrece un conjunto de opciones, una de las cuales es un submenú llamado Nuevo, con dos opciones: Test y Test Suite. Estas dos opciones permiten agregar al Test Suite seleccionado como padre un nuevo elemento vacío (Test o Test Suite), con un nombre por default, que podrá ser accedido de forma inmediata a través de la función de edición para proceder a la configuración del mismo.

8.2.2.5. Configuración de Tests y Test Suites

La configuración o edición de los elementos del árbol se realiza también desde el menú contextual que se despliega al cliquear con el botón derecho sobre alguno de ellos, mediante la opción Editar. Esta acción abre una de dos posibles ventanas, según se trate de un Test o de un TestSuite.

El caso de edición del Test Suite es relativamente simple puesto que así es su estructura; en este caso es posible editar sus descripciones corta y larga al desplegarse la siguiente

ventana:

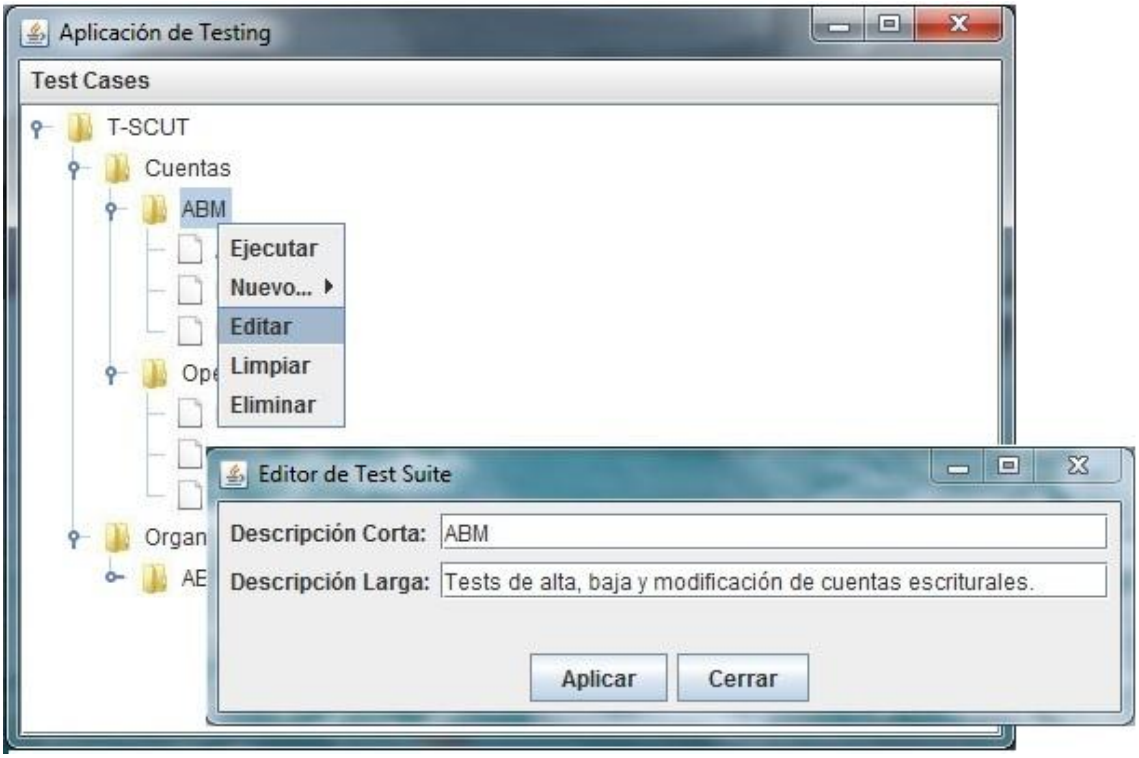


Figura 8.15 – Edición de un Test Suite

El caso de edición de un Test es mucho más complejo puesto que, como se ha visto en el modelo básico, éste conforma el corazón de tal modelo, por lo que la estructura a definir para el mismo es más compleja y mucho más dinámica. Al cliquear en la opción Editar se despliega la siguiente ventana:

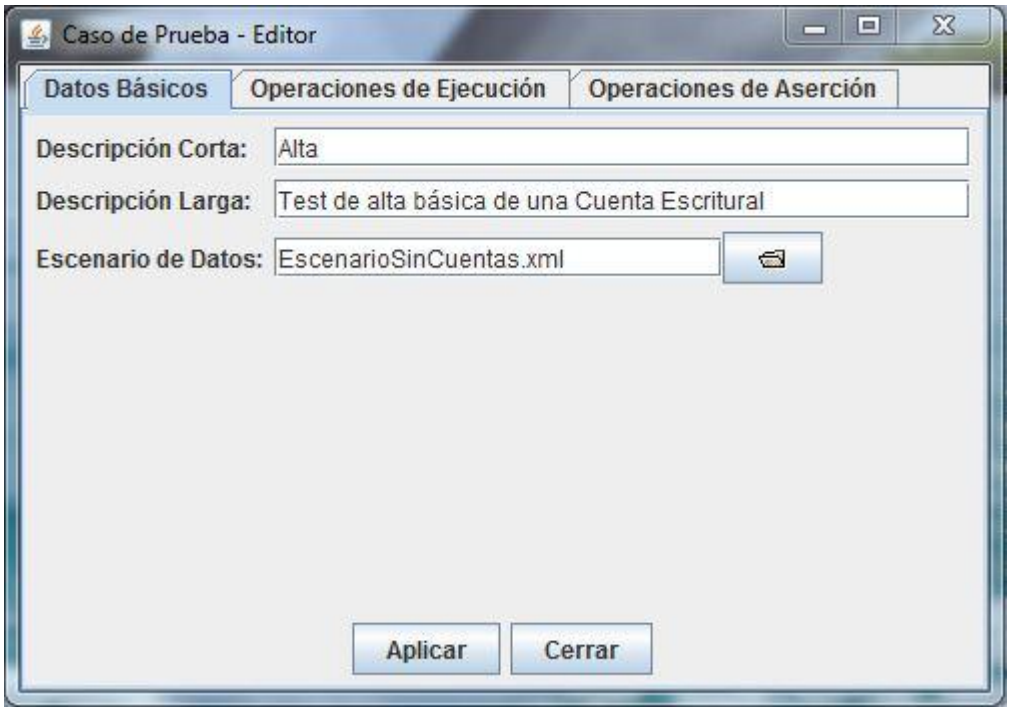


Figura 8.16 – Edición de un Test

Puede notarse que esta ventana divide la edición del test en tres partes fundamentales de su estructura, las cuales se reflejan a nivel interfaz en tres solapas diferentes: Datos Básicos, Operaciones de Ejecución y Operaciones de Aserción. Estas solapas se describen en las secciones siguientes.

8.2.2.5.1. Solapa Datos Básicos

En primera solapa se pueden editar las descripciones corta y larga del test como así también seleccionar el archivo .xml que contiene el juego de datos (dataset) de la aplicación objetivo que se cargará de forma automática e inmediata antes de la ejecución del test en cuestión. Esto último se realiza mediante la ventana de selección de archivo que se despliega al cliquear el botón del campo Escenario de Datos.

8.2.2.5.2. Solapa Operaciones de Ejecución

La segunda solapa, Operaciones de Ejecución, se muestra en la siguiente figura:

Figura 8.17 – Listado y detalle de las Operaciones de Ejecución de un test

Aquí se muestra el listado de operaciones de ejecución del test en la misma secuencia en que serán ejecutadas en el momento en que se corra el mismo. A su vez, al cliquear en una de tales operaciones, en la parte inferior de la ventana se mostrará la información básica de la misma.

Esta información descriptiva incluye:

- Grupo Funcional: es la descripción del servicio (objeto servicio) que provee la

operación seleccionada. Este nombre, "Grupo Funcional", parece adecuado para un usuario final quien no tiene la obligación ni formación para entender de servicios en una arquitectura SOA.

- Descripción de la Operación: Es la descripción larga de la operación seleccionada, la cual es provista por su Service Descriptor, el cual a su vez ha sido previamente importado desde un archivo .xml, tal como se describió anteriormente. Por otro lado, la descripción corta, es aquella que figura para esa operación justamente en el panel de selección de operaciones.
- Listado de Parámetros: aquí es en donde el dinamismo del modelo se refleja en la interfaz, ya que mostrará un listado de campos de entrada para los parámetros que espera recibir la operación. La cantidad y nombre de cada uno de ellos se obtienen de los descriptores cargados al modelo a partir del archivo .xml.

Por otro lado, estos campos informativos poseen tooltips que aparecen al posicionar el mouse sobre ellos. Mediante ellos se brinda mayor información sobre el campo en cuestión, como por ejemplo qué significa exactamente un parámetro. Tales ayudas por lo general corresponden a las descripciones largas de los campos, las cuales son provistas en los descriptores previamente importados desde un archivo .xml.

Por su parte, el botón Borrar permite eliminar la operación seleccionada, mientras que el botón Agregar Operación permite agregar a la lista una nueva operación de ejecución, lo cual se describe a continuación

8.2.2.5.2.1. Alta de Operaciones de Ejecución

El botón de alta de la segunda solapa despliega una nueva ventana, la cual se muestra a continuación:

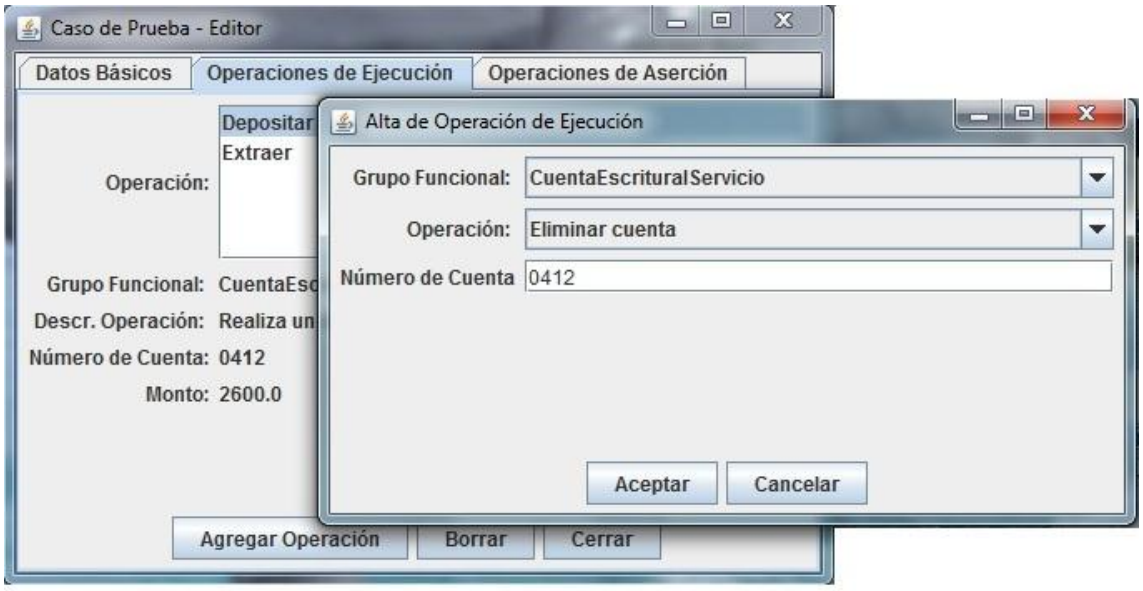


Figura 8.18 – Alta de Operaciones de Ejecución

En esta ventana debe seleccionarse, en primer lugar, el Grupo Funcional (servicio) al que pertenece la operación que se desea incorporar al test. Este selección actúa como filtro del

segundo listado, Operación, ya que éste listará las operaciones pertenecientes a al grupo funcional seleccionado.

Al seleccionar una operación en este segundo listado, la ventana mostrará dinámicamente campos de entrada para los parámetros que la operación seleccionada requiere, de forma tal que el usuario final especifique el valor concreto para cada uno de ellos según sean los fines del test que está configurando.

Para el caso en que el parámetro de entrada a especificar sea un DTO, la etiqueta del campo denotará el nombre de su clave funcional, de manera tal que el usuario final especifique su valor y, de forma transparente a este usuario, el sistema cree el DTO a partir del valor ingresado. Crear DTOs o enviar valores simples, son decisiones que el sistema puede tomar gracias a toda la información sobre parámetros y operaciones brindada por los descriptores. El usuario final jamás estará enterado de la existencia de DTOs, ya que se trata de detalles técnicos que no le competen.

En este último punto, en relación a los DTOs, se ha evaluado una alternativa de ofrecer al usuario el listado de DTOs según las entidades que ya tenga creadas la aplicación objetivo, de forma tal que el usuario simplemente se limite a seleccionar uno de ellos. Sin embargo, para lograr esto, la aplicación objetivo debería proveer servicios que permitieran obtener tales listados de DTOs. En esta línea, debería exigirse a tal aplicación la implementación de esos servicios, lo cual resultaría invasivo y contrarios a los lineamientos de flexibilidad establecidos para este trabajo. Es por esto que, cuando se trate de un DTO, se inducirá al usuario final a especificar su clave funcional, puesto que a partir de la misma se podrá crear y gestionar el DTO como parámetro de forma transparente y genérica, tal como se ha descripto en secciones anteriores.

Al presionar el botón Aceptar, la nueva operación será agregada al listado de operaciones de ejecución de la segunda solapa de la ventana de edición del test descripta previamente.

8.2.2.5.3. Solapa Operaciones de Aserción

La tercera solapa, Operaciones de Aserción, se muestra a continuación:

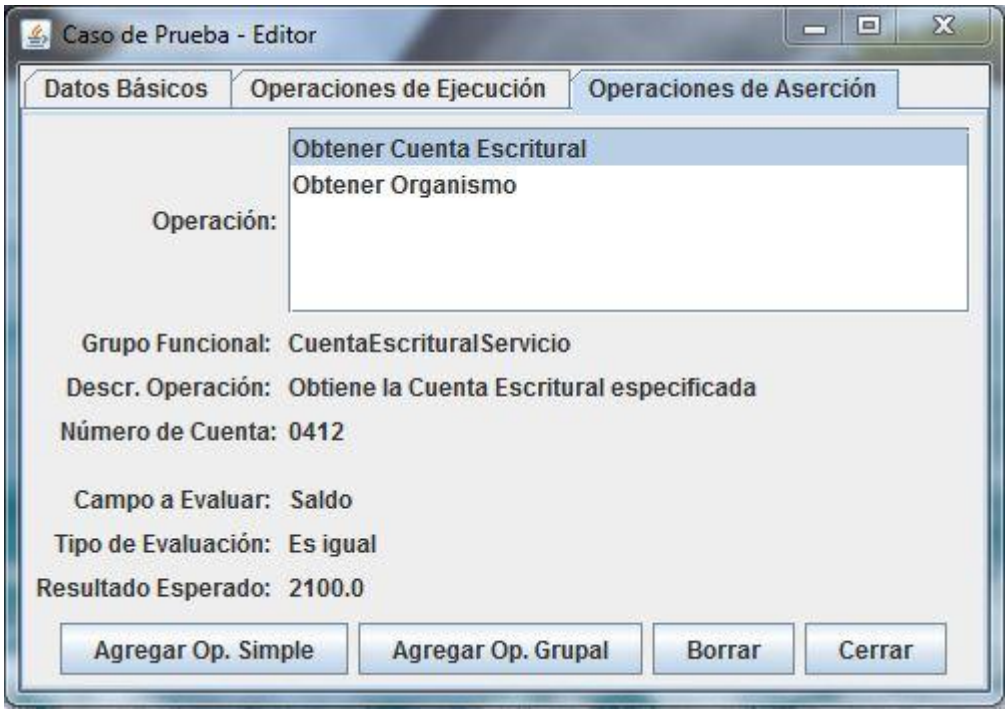


Figura 8.19 – Listado y detalle de las Operaciones de Aserción de un test

Aquí se muestra el listado de operaciones de aserción del test en la misma secuencia en que serán ejecutadas en el momento en que se corra el mismo. Esta solapa es muy similar a la segunda aunque un poco más compleja.

En primer lugar, este listado de operaciones lista de forma homogénea las operaciones de aserción por las que está compuesto el test, sean éstas de aserción simple (clase SingleTestAssertion del modelo básico) o de aserción grupales (GroupTestAssertion del modelo básico). Ambos tipos de operaciones pueden incluirse en un mismo listado. De la misma forma que en la degunda solapa, al seleccionar una operación se muestra debajo de ella información que la describe, la cual depende del tipo de operación que se trate (simple o grupal).

Si la operación seleccionada es simple (SingleTestAssertion), en primer lugar se muestran los parámetros y valores especificados por el usuario para la misma. Esto es igual que en la segunda solapa; a continuación se describen las novedades que incorpora esta tercera solapa.

Se muestra también un campo Tipo de Evaluación, el cual lista la estrategia de aserción (clase AssertionStrategy del modelo básico) que se ha elegido para realizar la aserción del test. Éstas se describen en términos comprensibles para el usuario final, por lo que pueden ser: Es Igual, Es Distinto, Nulo, No Nulo, Verdadero, Falso. Tal como se ha explicado en el modelo básico, estas estrategias pueden ser unarias o binarias, según sólo verifiquen el resultado obtenido o también un resultado esperado, respectivamente.

En el caso de que la estrategia sea binaria, se mostrará habilitado un campo Resultado Esperado, que muestra el valor especificado que se comparará con el resultado final obtenido de la operación de aserción según la estrategia de aserción especificada.

Por otro lado, si el resultado de la operación de aserción es un DTO (lo cual el sistema determina a partir de los descriptores importados), se ofrece al usuario el listado de formas de acceso a sus atributos (clase Accessor del modelo básico) a través de la lista desplegable etiquetada como Campo a Evaluar. Cabe destacar nuevamente que el usuario no entiende de DTOs, pero puede intuir fácilmente que un resultado está compuesto por varios datos y que se le permite el acceso a los mismos. Además, el usuario elige siempre un elemento de la lista, independientemente de que el mismo represente una sóla operación de acceso o una cadena de las mismas.

La información descriptiva explicada hasta aquí corresponde a las operaciones de aserción simples; para las operaciones de aserción grupales toda esta información no existe, ya que se trata simplemente de la comparación de un área de la base de datos obtenida como resultado contra el archivo que define el área de datos esperada. Es por esto último que como información descriptiva de las operaciones de aserción grupales se especifica esta condición junto con el nombre del archivo que contiene los datos esperados.

Finalmente, el botón Borrar permite la eliminación de la operación seleccionada, mientras que los botones Agregar Operación Simple y Agregar Operación grupal permiten agregar una operación de uno u otro tipo a la lista, la forma de lo cual se describe a continuación.

8.2.2.5.3.1. Alta de Operaciones de Aserción

La ventana de Alta de Operación Simple permite dar de alta una operación de aserción de ese tipo y es muy similar, aunque algo más compleja, a la ya descripta para Operaciones de Ejecución. Aquí también se muestra el listado de Grupos Funcionales que representan los servicios provistos por la aplicación objetivo y, para el Grupo Funcional seleccionado, se muestran las operaciones que éste implementa. En este caso, por tratarse de operaciones de aserción, se mostrarán sólo aquellas operaciones que devuelvan algún resultado, ya que es a partir del mismo que puede realizarse la aserción. Debajo de la operación seleccionada se muestra dinámicamente la lista de parámetros que define la misma, de forma tal que el usuario

final pueda especificar el valor para cada uno de ellos.

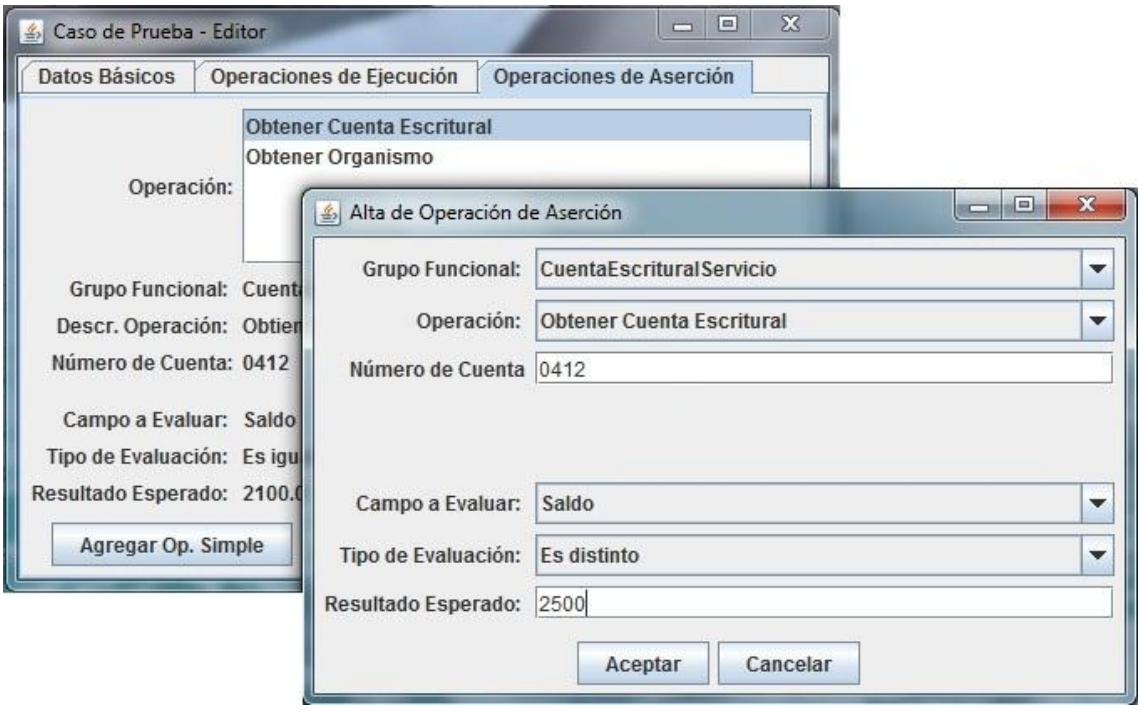


Figura 8.20 – Alta de Operaciones de Aserción Simples

A continuación también deben especificarse los campos Campo a Evaluar, Tipo de Evaluación y Resultado Esperado. El significado de estos campos ha sido explicado más arriba, aquí se agregará sólo que estos campos se habilitan e inhabilitan de forma dinámica ya que están interrelacionados. En este sentido, el campo Resultado Esperado se habilita sólo si la estrategia de aserción especificada en Tipo de Evaluación es binaria. Por otro lado, el campo Campo a Evaluar se habilita sólo si el resultado de la operación define accessors al resultado, en cuyo caso éstos se listan para que el usuario final pueda elegir uno de ellos.

Mediante el botón Aceptar se crea la operación de aserción simple y se agrega a la tercera solapa de la ventana de edición de tests ya descripta.

Por otro lado, la ventana de Alta de Operación Grupal, que se despliega presionando el botón correspondiente de la tercera solapa de la ventana de edición de tests, permite la especificación del área de base de datos de la aplicación objetivo que se desea que se considere en la aserción, como así también el archivo .xml que contiene los datos de esa área a los que se define como los esperados. Tal ventana se muestra a continuación:

Figura 8.21 – Alta de Operaciones de Aserción Grupales

Al presionar el botón Aceptar, se crea la operación de aserción grupal y se la agrega a la lista de operaciones de la tercera solapa de la edición de tests.

8.2.2.6. Ejecución de Tests

La ejecución de tests o test suites se lleva a cabo posicionándose sobre el nodo del árbol correspondiente y seleccionado la opción Ejecutar del menú contextual. En caso de tratarse de un TestSuite, esta acción ejecuta todo el subárbol del cual el nodo seleccionado es raíz.

Esta acción actualiza todos los íconos de éxito o falla del árbol tal como se ha descripto previamente.

8.2.2.7. Eliminación y Limpieza de Tests

Mediante la opción Eliminar del menú contextual puede eliminarse un test o un test suite, esto último elimina el subárbol completo que lo tiene como raíz.

Mediante la opción Limpiar se resetea el estado del test o test suite, esto significa que se lo pone en estado "No ejecutado" (sin ícono de éxito ni de falla). En el caso que se trate de un test suite, se resetea el estado de todo el subárbol que lo tiene como raíz.

Estas operaciones pueden generar un cambio de estado en los ancestros de los nodos en los cuales se opera. Como ejemplo, puede considerarse que la eliminación del único hijo fallado de un test suite cambia el estado de este último si el resto de sus hijos estaban en estado "Exitoso".

9. CONCLUSIONES Y TRABAJOS FUTUROS

Al igual que en una gran cantidad de sistemas informáticos de la industria, el Sistema Cuenta Única del Tesoro se basa en un modelo por capas, con tecnología de objetos Java en la del modelo, bases de datos Oracle en la de Persistencia y un mapeador Hibernate entre ambas. Asimismo, siguiendo los lineamientos de la Ingeniería de Software y de la industria, se realizan tests de unidad sobre la capa del modelo, para lo cual se utiliza JUnit.

Actualmente, en el proyecto, los juegos de datos de los tests de unidad son configurados y cargados al sistema por los desarrolladores. Esto tiene la desventaja de que esta tarea les insume demasiado tiempo y esfuerzo, restándoles tiempo efectivo para las tareas de desarrollo. Además de lo anterior, y por lo mismo, los juegos de datos generados son relativamente chicos, por lo que algunos tests pierden eficacia al no poder ser evaluados en contextos un poco más reales, como son algunos de mayor volumen de datos que se utilizan en este dominio financiero. Finalmente, los tests de unidad actuales mediante JUnit realizan aserciones sobre resultados puntuales, y no sobre áreas más grandes y completas del dominio para asegurar que ningún dato de esa área haya cambiado debido a efectos laterales o a la introducción accidental de nuevos errores.

Se entiende que DBUnit provee soluciones a las tres desventajas mencionadas, consideradas de importancia para el proyecto SCUT. En primer lugar, el equipo de desarrollo podría delegar en el usuario final la creación de los juegos de datos mediante las GUIs actuales; esto último tiene además la ventaja política, desde el punto de vista de la gestión del proyecto, de que el sistema se testea con los datos que el mismo usuario final ha cargado. Por otro lado, y dado que para la generación del juego de datos se utilizarían las interfaces existentes (y no programación Java), los mismos podrán ser de mayor tamaño. Por último, DBUnit provee aserciones que permiten comparar el resultado de una gran área de la Base de Datos con el estado esperado, asegurando que ningún dato ha sido alterado fuera de lo planificado.

No obstante las ventajas del uso de DBUnit mencionadas en el párrafo anterior, aún se necesita asistencia de programadores para la realización de los tests, siendo que por lo general tanto su motivación como necesidad del proyecto se enfocan hacia el desarrollo de nueva funcionalidad y mantenimiento de los sistemas en producción. En este sentido, este trabajo propone el desarrollo de una aplicación sobre la base provista por DBUnit y otras tecnologías de licencia libre, que permite al usuario final configurar y gestionar tests de regresión de manera amigable sin ningún tipo de intervención por parte de los programadores. Esta solución cuenta con la ventaja adicional de delegar las tareas de testing de regresión en estos usuarios finales, que son quienes realmente comprenden el dominio del problema y tienen la capacidad de realizar tests realmente significativos.

Frente a otras soluciones similares existentes en el mercado, la propuesta de este trabajo presenta las ventajas de que, por estar construida totalmente con software libre, el costo económico de la misma es nulo. Asimismo, por enfocarse en la capa de servicios y no directamente en las interfaces de usuario de la aplicación a testear, el testing que permite es cross-interfaces y, por lo mismo, no es sensible a pequeñas modificaciones que sufran estas interfaces. Por último, y también gracias a su foco sobre la capa de servicios, permite realizar tests de regresión de aplicaciones cuyas interfaces no son necesariamente gráficas.

Como trabajo futuro, el más importante es la adaptación y prueba de este esquema de testing para su uso en dos ambientes diferentes, es decir, uno para la aplicación de testing y otro para la aplicación objetivo. Esto es lo que permitirá finalmente llevar a nivel físico el desacoplamiento entre ambas aplicaciones que en este trabajo se ha logrado a nivel lógico.

Existen también varios desarrollos funcionales que permiten una extensión y mejora de las aplicaciones presentadas en este trabajo. Entre estas extensiones pueden citarse las siguientes:

- Una aplicación para la gestión de servicios. Esta aplicación (o módulo de la aplicación de testing) permitiría realizar la exportación e importación de servicios logrando mayor usabilidad, como por ejemplo recordando preferencias y configuraciones de usuario en cada exportación sucesiva. Actualmente cada exportación es totalmente nueva y se pierden todas las configuraciones anteriores realizadas por el usuario configurador.

- Un configurador y evaluador de expresiones booleanas, de forma de poder definir sentencias complejas pero amigables y dinámicas en la configuración de aserciones.

- Un módulo de métricas, estadísticas y reportes de tests.

10. FUENTES Y REFERENCIAS

1. [PMI] Project Management Institute (PMI): http://www.pmi.org/. Fecha de Acceso: 12/10/2015.

2. [IEEE] Institute of Electrical and Electronics Engineers (IEEE). "IEEE Standard Glossary of Software Engineering Terminology", IEEE Std. 610.121990. New York, NY: IEEE, 1990. https://www.ieee.org/. Fecha de Acceso: 12/10/2015.

3. [IEEE_UT] Institute of Electrical and Electronics Engineers (IEEE). "IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987", en IEEE Standards: Software Engineering, Volume Two: Process Standards; 1999 Edition.

4. [INTI] Instituto Nacional de Tecnología Industrial. "El Testing como parte del proceso de Calidad del Software". Material y Jornadas dictadas del 3/6/2013 al 5/6/2013, pág. 63.

5. [SEI] Software Engeneering Institute (SEI): http://www.sei.cmu.edu. Fecha de Acceso: 12/10/2015.

6. [McGregor] John D. McGregor, "Testing a Software Product Line": http://www.sei.cmu.edu/reports/01tr022.pdf, pág. 30.

7. [RalphJ] Don Roberts, Ralph Johnson, "Evolving frameworks: A pattern language for developing object-oriented frameworks". University of Illinois, Journal Article, Proceedings of the Third Conference on Pattern Languages and Programming.

8. [Gamma] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design patterns. Elements of reusable Object-Oriented Software. Adison Wesley, 1995. ISBN 0201633612.

9. [Rebecca] Object Design: Roles, Responsibilities, and Collaborations. Rebecca Wirfs-Brock, Alan McKean. ISBN 0201379430.

10. [PersOO] Persistencia orientada a objetos. Javier Bazzocco, Editorial de la Universidad Nacional de La Plata, 2012. ISBN 978-950-34-0821-6

11. [UML] "UML Gota a Gota", Martin Fowler, Kendall Scott. ISBN 9684443641.

12. [Ley13767] Ley de Administración Financiera. Nro. 13767 y reglamentación, decreto 3260/08. Provincia de Buenos Aires.

13. [JUnit] Sitio oficial del Producto. Especificaciones y descarga del software. http://junit.org/. Fecha de Acceso: 12/10/2015.

14. [DBUnit] http://dbunit.sourceforge.net. Sitio del producto, especificación, manuales y descarga del software. Fecha de Acceso: 12/10/2015.

15. [Hibernate] Sitio oficial del Producto. Especificaciones y descarga del software. http://www.hibernate.org Fecha de Acceso: 12/10/2015.

16. [MySQL] Sitio oficial del Producto. Manual de instalación, operación y download de "MySQL Community Server". http://www.mysql.com. Fecha de Acceso: 12/10/2015.

17. [EclipseSTS] Sitio official de Spring. Entorno de desarrollo Java con funcionalidad de

Spring. Especificación y descarga del producto. http://spring.io/tools. Fecha de Acceso: 12/10/2015.

18. [JDK] Java Development Kit. Especificación y kit de desarrollo Java verión jdk-6u32-windows-x64. http://www.oracle.com/technetwork/java/javase/downloads/index.html. Fecha de Acceso: 12/10/2015.

19. [Spring] Sitio Oficial de Spring. Especificación del framework de desarrollo Spring. http://spring.io/ . Fecha de Acceso: 12/10/2015.

20. [JAXB] Sitio de JAXB. JAXB Reference Implementation. https://jaxb.java.net/ Fecha de Acceso: 12/10/2015.

21. [Selenium-IDE] Especificación del Proyecto Selenium – IDE y descarga del software. . http://seleniumhq.org/ Fecha de Acceso: 08/10/2015.

22. [WebDriver] Especificación del Proyecto WebDriver y descarga del software. http://www.seleniumhq.org/projects/ide/ Fecha de Acceso: 10/10/2015.

23. [QALiber] Especificación de la herramienta QALiber y descarga del software. http://qaliber.org/ Fecha de Acceso: 13/10/2015.

11. ANEXO A: EJEMPLO DE USO BÁSICO DE DBUNIT

En esta sección se muestra un breve ejemplo de uso standard de DBUnit (para lo que fue concebido) con algunas de las clases de dominio utilizadas en el Sistema Cuenta Única del Tesoro. Las tecnologías utilizadas son exactamente las mismas (Java, JUnit, Hibernate) salvo por la Base de Datos Oracle del proyecto, que en estos ejemplos se reemplaza por una MySQL por razones legales de licencias. No obstante esto último, y dada la ya explicada transparencia en cuanto a la Capa de Persistencia que proveen tanto Hibernate como DBUnit, los ejemplos aquí expuestos se consideran pertinentes y representativos de la realidad del proyecto.

Todo el código y archivos de configuración utilizados en estos ejemplos se entregan como parte integrante del presente trabajo.

11.1. Recorte del Modelo a Utilizar

Las pruebas se realizarán con un recorte reducido del modelo financiero real cuyos aspectos muy generales han sido presentados en la sección 8.

A continuación se muestra el Modelo de Clases de Análisis (MCA) que se utilizará.

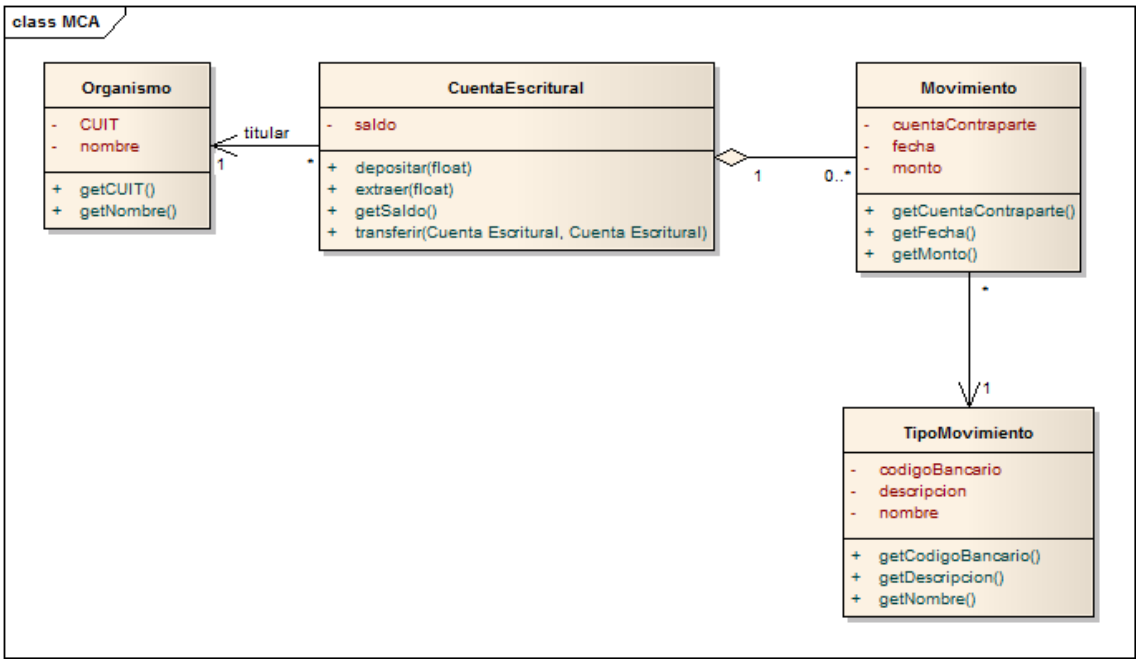


Figura A.1 – Recorte del modelo a utilizar como ejemplo

Puede observarse que cada Cuenta Escritural (CE), cuya naturaleza es netamente contable, pertenece a un Organismo del Estado, a quien se considera el tituar. Cada CE permitirá realizar tres tipos de operaciones: depósito, extracción y transferencia. Las cuentas mantendrán un registro de tales operaciones mediante un conjunto de movimientos consecutivos, que mantendrán sus datos correspondientes y su tipo (TipoMovimiento). Dado que en estas pruebas se realizarán operaciones simples, estos tipos de movimientos podrán ser de Crédito o Débito.

11.2. Modelo Java y Tests mediante JUnit

A continuación se muestra la implementación de la parte más relevante de la clase Cuenta

Escritural, perteneciente a la Capa del Modelo; las implementaciones de las demás clases forman parte de los entregables de este trabajo, por lo que no se reproducirán en esta sección.

```
package scut.model;

/**
 * Clase que representa a las cuentas escriturales del sistema financiero
 * provincial.
 * Estas cuentas mantienen su estado administrativo y financiero, entre
 * lo que se destaca su número, titular, saldo y los movimientos que han
 * ocurrido en la misma. A su vez, permiten las operaciones de depósito,
 * extracción y transferencia de fondos.
 *
 * @author dcano
 */
public class CuentaEscritural {

 protected long idCuentaEscritural;
 protected float saldo;
 protected Organismo titular;
 protected String numero;
 protected List<Movimiento> movimientos;


 public CuentaEscritural(String numero, Organismo titular) {
       super();
       this.numero = numero;
       this.titular = titular;
       this.setSaldo(0);
       this.setMovimientos((List<Movimiento>) new ArrayList<Movimiento>());
 }


 /**
  * Realiza el depósito de unMonto en la cuenta escritural receptora,
  * generando el asiento del movimiento correspondiente.
  *
  * @param unMonto
  */
 public void depositar(float unMonto) {
       this.setSaldo(this.getSaldo() + unMonto);
       Movimiento m = new Movimiento (null, new Date(), unMonto,
                     TipoMovimiento.getTipoMovimiento(667));
       this.agregarMovimiento(m);
 }


 /**
  * Realiza la extracción de unMonto en la cuenta escritural receptora,
  * generando el asiento del movimiento correspondiente.
  * La operación se realiza sólo si el saldo de la cuenta es mayor o
  * igual al monto a extraer.
  *
  * @param unMonto
  */
 public void extraer(float unMonto) {
       if (this.getSaldo() >= unMonto){
             this.setSaldo(this.getSaldo() - unMonto);
             Movimiento m = new Movimiento (null, new Date(), unMonto,
                           TipoMovimiento.getTipoMovimiento(800));
```

```
            this.agregarMovimiento(m);
      }
 }


 /**
  * Realiza la transferencia de unMonto desde la cuenta escritural
  * receptora hacia la cuentaEscrituralDestino, generando en ambas
  * cuentas los asientos de los movimientos correspondientes.
  * La operación se realiza sólo si el saldo de la cuenta receptora
  * es mayor o igual al monto a transferir.
  *
  * @param unMonto
  * @param cuentaEscrituralDestino
  */
 public void transferir(CuentaEscritural cuentaEscrituralDestino,
                        float unMonto) {
      if (this.getSaldo() >= unMonto){
            this.extraer(unMonto);
            cuentaEscrituralDestino.depositar(unMonto);
      }
 }

}
```

Figura A.2

A continuación vemos la clase que implementa tests de unidad para las instancias de la clase CuentaEscritural mostrada arriba. En particular, en el método setUp(), esta clase crea un mismo juego de datos que será re-cargado y utilizado en cada test ejecutado por la misma.

Los tres métodos de test testean las tres operaciones básicas de la cuenta: depósito, extracción y transferencia. En este documento sólo se muestra el test del depósito por razones de espacio y legibilidad.

Al final de cada uno se realizan las aserciones pertinentes para asegurar que los resultados de estas operaciones son los esperados.

```
package scut.test;

/**
* Test de Unidad de la Cuenta Escritural.
*
* Este test (alternativa 1) se basa exclusivamente en el test de unidad
* estandar mediante JUnit (no se utiliza DBUnit). Esto significa que el
* juego de datos a utilizarse es creado manualmente por un desarrollador,
* y mantenido siempre en memoria principal (no se utiliza una BD).
* Esto último puede apreciarse en el método setUp().
*
**/
public class CuentaEscrituralTest1 extends TestCase {

 Organismo o1, o2;
 CuentaEscritural ce1, ce2;
 TipoMovimiento tm1, tm2;
 private static SessionFactory sessions;


 public CuentaEscrituralTest1(String name) {
      super(name);
      // Configuración de Hibernate
```

```
        Configuration cfg = new Configuration();
        cfg.configure();
        sessions = cfg.buildSessionFactory();
 }


/**
 * Crea el juego de datos a utilizarse durante cada test.
 *
 * IMPORTANTE: Notar que en este setUP todos los objetos son creados de
 * forma manual por el programador y mantenido en memoria en todo momento
 * de cada test.
 **/
 public void setUp(){

        // Creación de Organismos.
        o1 = new Organismo("Asuntos Agrarios", "24-26563698-9");
        o2 = new Organismo("ARBA", "37-26563698-1");

        // Creación de Cuentas Escriturales.
        ce1 = new CuentaEscritural("0412",o1);
        ce2 = new CuentaEscritural("0229",o2);

        // Creación de Tipos de Movimientos.
        TipoMovimiento tm1 = new TipoMovimiento(667, "Ingreso",
                                                "Ingreso de fondos");
        TipoMovimiento tm2 = new TipoMovimiento(800, "Egreso",
                                                "Egreso de fondos");
        TipoMovimiento.addTipoMovimiento(tm1);
        TipoMovimiento.addTipoMovimiento(tm2);
 }

/**
 * Testea el caso básico del depósito en cuenta escritural.
 **/
 public void testDepositar(){

        ce1.depositar(100f);

        // Chequea que el saldo resultante de la cuenta sea correcto.
        assertEquals(ce1.getSaldo(),100f);

        // Chequea que la cantidad de movimientos generados sea correcta.
        int cantidadMovimientos = ce1.getMovimientos().size();
        assertEquals(cantidadMovimientos,1);

        // Chequea que el tipo de movimiento generado para el depósito
        // sea correcto.
        Movimiento movimiento=ce1.getMovimientos().get(cantidadMovimientos-
1);
        assertEquals(movimiento.getTipoMovimiento().getCodigoBancario(),
667);
 }
}
```

Figura A.3

Aquí puede observarse cómo el juego de datos (en el método setUp) debe ser programado manualmente por el desarrollador. También queda a la vista que estos datos residen en memoria, sin contemplar la capa de persistencia. Debido a las dos razones anteriores, estos juegos de datos suelen ser pequeños.

11.3. Tests mediante DBUnit

Aquí se muestra como la misma clase Cuenta Escritural se testea a partir de un juego de datos residente en la Capa de Persistencia. Nuevamente, sólo por cuestiones de espacio y legibilidad se muestra un solo método de test (depositar).

```
package scut.test;

/**
* Test de Unidad de la Cuenta Escritural.
*
* En este test (alternativa 2) se incluye la utilización de una base de
* datos mediante DBUnit.
* El juego de datos a utilizar (dataset) en cada test reside en el archivo
* "fullDataset.xml" y es automáticamente cargado en una base de datos
* automáticamente limpiada antes de cada ejecución.
*
* De esta manera, a diferencia de la alternativa 1, el desarrollador no
* necesita crear el juego de datos manualmente, sino solo levantar de la BD
* aquellos datos que desea incluir en los tests. Esto último puede apreciarse
* en el método loadSetUp().
*
* Una vez cargado el juego de datos, las etapas de ejecución y asserción de
* los tests son exactamente iguales a las de la alternativa 1.
*
**/
public class CuentaEscrituralTest2 extends DBTestCase {

    Organismo o1, o2;
    CuentaEscritural ce1, ce2;
    TipoMovimiento tm1, tm2;

    private static SessionFactory sessions;

    public CuentaEscrituralTest2(String name) {
       super(name);
       // Conexión con la BD.
        System.setProperty(
        PropertesBasedJdbcDatabaseTester.
                                        DBUNIT_DRIVER_CLASS,"com.mysql.jdbc.Driver" );
       System.setProperty( PropertiesBasedJdbcDatabaseTester.
                                        DBUNIT_CONNECTION_URL,"jdbc:mysql://localhost/test"
                                    );
       System.setProperty( PropertiesBasedJdbcDatabaseTester.
                                        DBUNIT_USERNAME, "root" );
       System.setProperty( PropertiesBasedJdbcDatabaseTester.
                                        DBUNIT_PASSWORD, "" );
       Configuration cfg = new Configuration();
       cfg.configure();
       sessions = cfg.buildSessionFactory();
    }
```

```
 /**
  * Obtiene el archivo XML con los datos a cargar en la base de datos
  * antes de la ejecución de cada test.
  */
 protected IDataSet getDataSet() throws Exception {

        //Nombre del archivo Dataset a levantar.
        String fileName = "fullDataset.xml";
        FileInputStream fis = null;
        try{
                fis = new FileInputStream(fileName);

        } catch(Exception e){
                System.out.println("Error en carga de " + fileName);
                e.printStackTrace();
        }

   return new FlatXmlDataSetBuilder().build(fis);
}


/**
 * Levanta desde la base de datos (previemente cargada con los datos
 * del archivo XML dataset) los objetos puntuales del juego de datos
 * a partir de los cuales se disparan los tests.
 *
 * IMPORTANTE: Notar que en este setUP todos los objetos se levantan
 * desde la base de datos: el programador no tiene la necesidad de
 * crealos manualmente (ni siquiera de conocer al detalle la composición
 * de los mismos).
 **/
 public void loadSetUp(){

        Session session = sessions.openSession();
        Transaction tx = null;
        try {
                tx = session.beginTransaction();
                tx.begin();

                // Carga de los Tipos de Movimientos.
                Query query1 = session.createQuery(
                                "select tm from scut.model.TipoMovimiento tm");
                TipoMovimiento.setTipoMovimientos(query1.list());

                // Carga de las Cuentas Escriturales a utilizar.
                Query query2 = session.createQuery(
                                "select ce from scut.model.CuentaEscritural ce
                                  where ce.numero = '0412'");
                Query query3 = session.createQuery(
                                "select ce from scut.model.CuentaEscritural ce
                                  where ce.numero = '0229'");
                ce1 =   (CuentaEscritural) query2.list().get(0);
                ce2 =   (CuentaEscritural) query3.list().get(0);
```

```
                tx.commit();
                session.flush();

        } catch (Exception e) {
                e.printStackTrace();
                if (tx != null)
                        tx.rollback();
                session.close();
        }
        session.disconnect();
}


/**
 * Testea el caso básico del depósito en cuenta escritural.
 **/
public void testDepositar(){
        loadSetUp();

        ce1.depositar(100f);

        // Chequea que el saldo resultante de la cuenta sea correcto.
        assertEquals(ce1.getSaldo(),100f);

        // Chequea que la cantidad de movimientos generados sea correcta.
        int cantidadMovimientos = ce1.getMovimientos().size();
        assertEquals(cantidadMovimientos,1);

        // Chequea que el tipo de movimiento generado para el depósito sea correcto.
        Movimiento movimiento = ce1.getMovimientos().get(cantidadMovimientos - 1);
        assertEquals(movimiento.getTipoMovimiento().getCodigoBancario(), 667);
}
```

Figura A.4

A diferencia del caso anterior, a través del método de set up (loadSetUp()), el programador no tendrá la necesidad de crear el juego de datos manualmente, sino que sólo se limitará a poner disponibles para el test algunos objetos que ya residen en la Capa de Persistencia. Estos objetos persistentes pertenecen al juego de datos creado por el usuario final, por lo que puede notarse que, a diferencia del ejemplo anterior, el desarrollador se limita a traerlos a memoria, librándose de la tarea de especificar valores concretos para cada uno de sus atributos. Esto es, no necesita conocer al detalle la composición interna de los objetos que conforman el juego de datos del test de unidad.

En cuanto a las aserciones, pueden observarse que son idénticas a las de la sección anterior.

11.4. Tests mediante DBUnit – Aserciones de Grupo.

En el ejemplo anterior se describió cómo un test mediante DBUnit puede levantar el juegos de datos de una base de datos y, luego de la ejecución, realizar aserciones sobre el

estado de algunos objetos puntuales. Sin embargo, como se dijo anteriormente, además de aserciones puntuales DBUnit permite chequear el estado de grandes áreas de la base de datos para asegurar que ningún otro dato ha cambiado conforme a lo esperado (y no sólo los valores puntuales chequeados). A continuación se muestra la aserción que permite realizar esto.

```
/**
* Test de Unidad de la Cuenta Escritural.
*
* Al igual que en la alternativa 2, en este test (alternativa 3), se incluye
* la utilización de una base de datos mediante DBUnit. De la misma manera,
* el juego de datos a utilizar (dataset) en cada test reside en el archivo
* "fullDataset.xml" y es automáticamente cargado en una base de datos
* automáticamente limpiada antes de cada ejecución.
*
* Aquí el desarrollador tampoco necesita crear el juego de datos manualmente,
* sino solo levantar de la BD aquellos datos que desea incluir en los tests.
* Esto último puede apreciarse en el método loadSetUp().
*
* La diferencia básica con la alternativa 2 es que aquí no sólo se realizan
* aserciones sobre datos y objetos puntuales, sino también sobre una gran área
* de la base de datos. Esto permite asegurarse de que no sólo algunos
* resultados puntuales son correctos, sino también de que no se producen
* efectos laterales en el resto de los datos del modelo.
*
**/
public class CuentaEscrituralTest3 extends DBTestCase {

    Organismo o1, o2;
    CuentaEscritural ce1, ce2;
    TipoMovimiento tm1, tm2;
    Session session;

    private static SessionFactory sessions;

    public CuentaEscrituralTest3(String name) {
      super(name);
      //Comienzo de Inicialización.
      System.setProperty( PropertiesBasedJdbcDatabaseTester.
                          DBUNIT_DRIVER_CLASS, "com.mysql.jdbc.Driver" );
      System.setProperty( PropertiesBasedJdbcDatabaseTester.
                          DBUNIT_CONNECTION_URL, "jdbc:mysql://localhost/test" );
      System.setProperty( PropertiesBasedJdbcDatabaseTester.
                          DBUNIT_USERNAME, "root" );
      System.setProperty( PropertiesBasedJdbcDatabaseTester.
                          DBUNIT_PASSWORD, "" );
      Configuration cfg = new Configuration();
      cfg.configure();
      sessions = cfg.buildSessionFactory();
    }

    /**
     * Obtiene el archivo XML con los datos a cargar en la base de datos
     * antes de la ejecución de cada test.
     */
    protected IDataSet getDataSet() throws Exception {
```

```
            //Nombre del archivo Dataset a levantar.
            String fileName = "fullDataset.xml";
            FileInputStream fis = null;
            try{
                    fis = new FileInputStream(fileName);
            } catch(Exception e){
                    System.out.println("Error en carga de " + fileName);
                    e.printStackTrace();
            }

      return new FlatXmlDataSetBuilder().build(fis);
   }

  /**
   * Levanta el juego de datos a utilizarse durante cada test.
   *
   * IMPORTANTE: Notar que en este setUP todos los objetos se levantan
   * desde la base de datos: el programador no tiene la necesidad de
   * crealos manualmente.
   **/
   public void loadSetUp(){

                    // Carga de los Tipos de Movimientos.
                    Query query1 = session.createQuery(
                                            "select tm from scut.model.TipoMovimiento tm");
                    TipoMovimiento.setTipoMovimientos(query1.list());

                    // Carga de las Cuentas Escriturales.
                    Query query2 = session.createQuery(
                                            "select ce from scut.model.CuentaEscritural ce
                                             where ce.numero = '0412'");
                    Query query3 = session.createQuery(
                                            "select ce from scut.model.CuentaEscritural ce
                                             where ce.numero = '0229'");
                    ce1 =    (CuentaEscritural) query2.list().get(0);
                    ce2 =    (CuentaEscritural) query3.list().get(0);
   }


  /**
   * Testea el caso básico del depósito en cuenta escritural.
   **/
   public void testDepositar() throws Exception{

            session = sessions.openSession();
            Transaction tx = null;
            try {
                    tx = session.beginTransaction();
                    tx.begin();

                    // Carga desde la base de datos el juego de datos del test.
                    loadSetUp();

                    // Realiza las operaciones propias del test (en este caso, el
```

```
                    // depósito).
                    ce1.depositar(100f);

                    tx.commit();
                    session.flush();
                } catch (Exception e) {
                    e.printStackTrace();
                    if (tx != null)
                        tx.rollback();
                    session.close();
                }
                session.disconnect();

                // Chequea que el saldo resultante de la cuenta sea correcto.
                assertEquals(ce1.getSaldo(),100f);

                // Chequea que la cantidad de movimientos generados sea correcta.
                int cantidadMovimientos = ce1.getMovimientos().size();
                assertEquals(cantidadMovimientos,1);

                // Chequea que el tipo de movimiento generado para el depósito
                // sea correcto.
                Movimiento movimiento = ce1.getMovimientos().get(cantidadMovimientos - 1);
                assertEquals(movimiento.getTipoMovimiento().getCodigoBancario(), 667);

                // A continuación chequea que el estado de todo el resto de las
                // cuentas escriturales del sistema sea el esperado. Con esto se
                // busca eliminar la posibilidad de efectos laterales en otras cuentas.

                // Levanta los datos de la base de datos luego de ejecutar el test.
                // En este caso, levanta todas las Cuentas Escriturales.
                IDataSet databaseDataSet = getConnection().createDataSet();
                ITable actualTable = databaseDataSet.getTable("CuentaEscritural");

                 // Levanta el juego de datos esperado (expected dataset), que se
                // encuentra provisto en un archivo XML (expectedDataset.xml).
                IDataSet expectedDataSet = new FlatXmlDataSetBuilder().build(
                                                    new File("expectedDataset.xml"));
                ITable expectedTable = expectedDataSet.getTable("CuentaEscritural");

                 // Realiza el assert de que el resultado obtenido del estado de
                 // TODAS las cuentas escriturales del sistema sea el esperado.
                Assertion.assertEquals(expectedTable, actualTable);
    }
}
```

Figura A.5

Este test nos asegura que ninguna otra cuenta escritural ha cambiado como efecto lateral de la operación en las dos cuentas ejercitadas en los tests.

11.5. Generación del Juego de Datos

Antes de comenzar la implementación de los casos de tests, los usuarios finales deben dejar la base de datos (de un ambiente de pruebas, por supuesto) en un estado al que consideren inicial, representativo y sobre el cual se pueda ejecutar cada uno de los tests una y

otra vez. Estando la base en este estado, el desarrollador sólo debe ejecutar el siguiente código para obtener un XML de la base de datos completa. Existen otras opciones que permiten extracciones parciales, más fáciles de manejar.

```
public class DatabaseXMLExport {

  public static void main(String[] args) throws Exception
    {
        // Conexión a la BD.
        Class driverClass = Class.forName("com.mysql.jdbc.Driver");
        Connection jdbcConnection = DriverManager.getConnection(
                          "jdbc:mysql://localhost/test", "root", "");
         IDatabaseConnection connection = new
                           DatabaseConnection(jdbcConnection);

        // Export full de la BD.
        IDataSet fullDataSet = connection.createDataSet();
        FlatXmlDataSet.write(fullDataSet,
                            new FileOutputStream("fullDataset.xml"));

        System.out.println("- Export del XML de la BD completado -
");
    }

}
```

Figura A.6

A continuación se muestra un ejemplo muy pequeño del XML generado a partir de la bases de datos entregada.

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>

      <CuentaEscritural
            idCuentaEscritural="196608"
            saldo="0.0"
            numero="0412"
            idOrganismo="131072"/>

       <CuentaEscritural
            idCuentaEscritural="196609"
            saldo="0.0"
            numero="0229"
            idOrganismo="131073"/>
</dataset>
```

Figura A.7

También es posible obtener el archivo DTD que define la estructura del archivos XML anterior (que a su vez es la del modelo de base de datos), el cuál se utiliza para realizar verificaciones sobre la validez de éste último. La forma de obtención es la siguiente:

```
public class DatabaseDTDExport {

    public static void main(String[] args) throws Exception{
```

```
        // Conexión a la BD.
        Class driverClass =
    Class.forName("com.mysql.jdbc.Driver");
        Connection jdbcConnection = DriverManager.getConnection(
                        "jdbc:mysql://localhost/test", "root", "");
        IDatabaseConnection connection =
                        new DatabaseConnection(jdbcConnection);

        // Escritura del archivo DTD.
        FlatDtdDataSet.write(connection.createDataSet(),
                            new FileOutputStream("test.dtd"));

        System.out.println("-Export del DTD de la BD completado-
    ");
    }
}
```

Figura A.8

En la entrega realizada puede verse el archivo DTD generado mediante este código.

12. ANEXO B: INSTALACIÓN DEL SOFTWARE ENTREGADO

En esta sección se describen las componentes del software entregado y una guía rápida de instalación y ejecución del mismo.

12.1. Componentes del Software

El software entregado está compuesto por las siguientes partes:

1) MySQL Community Server [MySQL]: Se trata de una versión de MySQL no comercial, pero con todas las características necesarias para los requerimientos de este trabajo.

2) Proyecto Java. Incluye los modelos y aplicaciones desarrolladas.

3) JUnit 4.11.

4) DBUnit 2.4.9.

5) Hibernate 3.

6) Driver JDBC para conexión a MySQL.

7) Archivo de juego de datos de prueba (dataset.xml) para la ejecución de tests.

8) Archivos de descriptores de servicios de la aplicación objetivo (serviceDescriptors.xml), para poder realizar pruebas de configuración y ejecución de tests.

12.2. Guía de Instalación

A continuación se describen los pasos para la instalación de los componentes anteriores, lo cual permite tener el proyecto funcionando y listo para la ejecución de las aplicaciones correspondientes.

1) Instalación del Servidor de Base de Datos MySQL (MySQL Community Server).

   - La instalación es muy sencilla y además en la próxima sección se listan algunos comandos básicos que pueden resultar de utilidad. De ser necesario, el manual de instalación y operación se encuentra on-line [MySQL].
   - El servidor de BD puede contener varias BDs. Para el proyecto entregado se utiliza la base de datos llamada “test” y el usuario utilizado es “root”, sin contraseña.

2) Creación del Proyecto Java.

   - Puede realizarse de forma sencilla mediante Eclipse STS (Spring Tool Suite) [EclipseSTS]. El proyecto java entregado ya está configurado con las siguientes partes: Hibernate, JUnit, DBUnit y el driver JDBC para MySQL, con lo cual no es necesario instalar ni configurar estas partes por separado.

3) Carga inicial de la BD.

   - Permite realizar la creación del modelo de la Base de Datos y la carga inicial de

datos. Para esto, con el servidor de BD levantado, sólo debe ejecutarse la clase testing.database.CreateSchema.java.

4) Ejecución de las aplicaciones

- La Aplicación Objetivo se ejecuta desde la clase scut.application.SCUTApplication, mientras que la aplicación de testing se ejecuta desde la clase testing.application.TestingApplication.

12.3. Comandos básicos de Operación de la Base de Datos MySQL.

A continuación se presenta una breve lista con los comandos básicos para la inicialización, operación y shutdown de la base de datos.

- Arranque del Servidor de BD:
  - **Desde una consola de commandos del sistema operativo: ...\MySQL Server 5.5\bin\mysqld --console**
- Bajada del Servidor de BD:
  - **Desde una consola de commandos del sistema operativo: ...\MySQL Server 5.5\bin\mysqladmin -u root shutdown**
- Conexión al servidor de BD:
  - **Desde una consola de commandos del sistema operativo: ...\MySQL Server 5.5\bin\mysql –u root**
- Desconexión del servidor de BD:
  - shell> **quit; (el shell queda habilitado en la misma ventana luego de hacer el "start").**
- Listado de Bases de Datos residentes en el Servidor:
  - shell> **show databases;**
- Conexión a una BD determinada del Servidor:
  - shell> **use *nombre_BD;* (ejemplo: use temp;)**
- Listado/descripción de tablas de la BD en uso (a la que se está conectado):
  - shell> **show tables; (muestra todas las tablas del servidor).**
  - shell> **describe table_name; (muestra la estructura de la tabla table_name).**
- Borrado de una tupla:
  - mysql> Delete from table_name where…;
- Selección de tuplas:
  - mysql> Select * from table_name where…;