Universidad de las Ciencias Informáticas

Facultad 4

Pruebas Unitarias de Software en la Plataforma J2EE

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor: Franklin Rivero Duharte

Tutor: Ing. Madelín Haro Pérez

Ciudad de La Habana, Junio de 2008

DECLARACIÓN DE AUTORÍA.

Declaro ser autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de _____ del año 2008

_____	_____
Firma del Autor	Firma del Tutor
Franklin Rivero Duharte	**Ing. Madelín Haro Pérez**

DATOS DE CONTACTO DEL TUTOR.

Graduada en el 2002 de la carrera Ingeniería Informática en el ISPJAE.

Ha trabajado en la Empresa de Servicios Informáticos (ESI) de Cienfuegos y a partir de enero de 2003 en la UCI.

Ha sido tutora de tesis durante los 5 cursos de la UCI y cotutor o consultor. Los temas trabajados han estado relacionados con el ciclo de vida del software (análisis, base de datos, arquitectura, prototipos, requisitos), Gestión de Proyecto, Seguridad de sistemas informáticos y Linux.

Ha participado en eventos como Fórum de Ciencia y Técnica, UCIENCIA y Universidad 2008.

Opta actualmente por la categoría docente de asistente y es integrante en la maestría de Gestión de Proyecto.

Se ha vinculado a los proyectos productivos en el rol de Jefe de Proyecto, Analista y Jefe de Capacitación

Ha impartido cursos de postgrado a funcionarios de empresas, profesores y trabajadores de la UCI y a los Directores de los IPI.

Ha impartido cursos de capacitación en dos proyectos de producción en el extranjero.

Ha recibido curso de postgrado en el instituto TATA en la India recibiendo el certificado internacional de habilidades.

Es asesora del Departamento Docente Central de Práctica Profesional.

AGRADECIMIENTOS.

En lo familiar,

A mi mamá por todos estos años de esfuerzo, sacrificio y dedicación para que yo haya podido tener y ser todo lo que hasta hoy he logrado. A todas mis tías que siempre me han demostrado mucho cariño, en especial a Juana, por brindarme su amor incondicional todos estos años que he compartido con ella. A mis primos por las maldades, a los grandes, a los chiquitos, a los de mi tiempo. A toda mi familia, por ser parte importante en mi vida y sentirse siempre orgullosos de mí. A mi hermana, mi cuñado y mi sobrina, por marcar momentos felices en mi vida. A mi abuelo.

En lo amistoso,

A todos mis amigos, de Santiago y de la UCI, a mis compañeros del grupo 2205 y los del proyecto SIGEP, y otros que he adoptado. A los que siempre han tenido una mano para ayudar y un rato para compartir. Esos que sin dudas son los mejores que he tenido, los que más he querido y los que extrañaré muchísimo: a Angelito y Elitico, los de siempre; a los prisioneros, los únicos (son muchos y cada uno especial); a Daylí, por ser la más dedicada de todas.

En lo profesional,

A mis profesores de todas las etapas de vida escolar, todos muy buenos y que tanto me han enseñado. A esos que son la motivación a ser mejor cada día, y son el impulso para transmitir todo lo que de ellos he obtenido: a Iosev y Luis Alberto, por confiar en mi voluntad; a Lester, Ramón Ernesto y Arturo, por demostrar ser buenos profesionales; a Américo por su consagración y cariño.

En la tesis,

A Made, mi tutora, por toda la ayuda y soportarme todos estos años. Por ser amiga y tan preocupada con todos nosotros; por hacerme ingeniero.

En lo personal,

A Leuken, por ser mi vida. Por ser mi amiga, mi compañera, mi confidente. La persona que me ha permitido quererle sin reservas y que me ha hecho completamente feliz. El impulso diario de todas mis acciones. Mi M., mi N., mi E., mi C.R.

DEDICATORIA.

A todos los que de una manera u otra se sientan con deseos de aprender y de ser buenos en lo que hacen. Que encuentren en este trabajo, parte de aquello que los hagan buenos y mejores profesionales.

RESUMEN.

Numerosos proyectos en la UCI son creados con personal inexperto en técnicas avanzadas de programación, por lo que no toman en cuenta las pruebas unitarias como una actividad fundamental en el proceso de desarrollo del software. El presente trabajo es una recopilación de información referente a las pruebas unitarias en la plataforma J2EE con el fin de apoyar y facilitar la labor de los desarrolladores durante el proceso de construcción de un sistema informático y se describen un grupo de pasos a tener en cuenta para su creación y ejecución con un costo y una calidad permisible.

Se indica la importancia y trascendencia de las pruebas unitarias. Se exponen sus principales características y propósitos, las ventajas y las desventajas que reportan, las diferentes formas en que pueden ser concebidas, así como los procedimientos que pueden tenerse en cuenta para realizarlas.

Se consideró la bibliografía consultada y la experiencia del autor en el trabajo con la plataforma J2EE y las herramientas que esta proporciona.

PALABRAS CLAVE.

Pruebas unitarias de software, herramientas de pruebas automatizadas, técnicas de pruebas unitarias, pruebas unitarias en J2EE, clases de prueba, métodos de prueba, cobertura de código, JUnit.

ÍNDICE

INTRODUCCIÓN.

En el desarrollo de software se hace necesaria la constante revisión y corrección de artefactos que se generan durante el proceso. Se realizan actividades que procuran establecer un mínimo de calidad. Se verifica que los entregables al cliente cuenten con los requisitos necesarios que se exigen y que se cumple con los estándares asumidos en el proyecto. Lo más importante para los integrantes del equipo de desarrollo es crear un producto que sea eficaz y eficiente.

Las pruebas a los componentes de software construidos durante la etapa de implementación cobran gran importancia en este aspecto pues se centran en encontrar y documentar defectos en la aplicación. Son las encargadas de velar por el cumplimiento de los requerimientos del sistema. Contribuyen a mejorar la productividad de los desarrolladores y la calidad final del producto. Además, la automatización del proceso de pruebas, hace que se gane en tiempo y se introduzcan menos errores humanos en tales revisiones.

Para destacar la importancia que tienen las pruebas automáticas de software en el proceso de desarrollo, Rob Johnson dijo: “Una revolución ha ocurrido en el desarrollo del software en los últimos años. De ser un marginado del desarrollo, despreciado por esos con más cosas importantes e interesantes que hacer, las pruebas – al menos, las pruebas unitarias – se han convertido en el corazón del proceso de desarrollo. Finalmente, en vez de ser visto como un requerimiento tedioso de ciertos procesos (…), ha sido integrado como una manera altamente productiva de trabajo.”(JOHNSON, ROD and HOELLER 2004)

En la Universidad de las Ciencias Informáticas (UCI), la mayoría de los proyectos son creados con estudiantes inexpertos en desarrollo de aplicaciones empresariales, no dominan la plataforma en la que desarrollarán el sistema informático y cuentan con un período limitado de producción. Son adiestrados en corto tiempo y adquieren conocimientos básicos del contenido necesario para comenzar a desarrollar en una etapa inmediata. Con esta situación, en muchos casos, pasa a un plano degradado el entrenamiento en técnicas de pruebas automáticas de software. Además, es insuficiente la preparación en temas relacionados con buenas prácticas de programación, uso de patrones y otras cuestiones técnicas, que les permita escribir código que sea viable en su prueba.

En estas condiciones, los desarrolladores depuran de manera manual el código que escriben empleando mucho tiempo por lo engorroso del proceso de recrear condiciones apropiadas. La mayor preocupación reside en corregir el código y no en implementar las funcionalidades necesarias. En muchos casos se detectan los errores que implican cambios de manera tardía, provocando una regresión en lo que se había empleado tiempo en implementar y retrasos en los planes del desarrollo del proyecto. En la corrección de los errores no se asegura poder evitar que se inserten otros nuevos.

No se garantiza un mínimo de calidad requerida cuando la revisión está sujeta a mecanismos intuitivos y criterios humanos.

Entonces, *¿cómo aplicar técnicas y buenas prácticas en el desarrollo de pruebas unitarias automatizadas, para un sistema basado en J2EE, con las herramientas que la plataforma posee?*

El problema científico planteado, enmarca su objeto de estudio en *las pruebas automatizadas de software*, teniendo como campo de acción *las pruebas unitarias de software en la plataforma J2EE*. El objetivo general es *capacitar a los desarrolladores en los pasos, técnicas y buenas prácticas a seguir en la realización de pruebas unitarias de softwares construidos en J2EE, utilizando las herramientas de la plataforma*.

Con el conocimiento de técnicas de pruebas automatizadas de software y el dominio de las herramientas adecuadas, los desarrolladores pueden plantearse estrategias en las pruebas. Su uso se vuelve más efectivo, ahorrando tiempo y recursos en las revisiones y detecciones de errores en el sistema. El equipo de desarrollo puede conducirse a un avance continuo e ininterrumpido en las metas propuestas por el proyecto. Se procura un mínimo de calidad en los componentes que se implementan y se obtiene en gran medida el producto deseado.

Para el cumplimiento del objetivo planteado se realizarán las siguientes tareas:

- Describir métodos generales que proponen diferentes metodologías de desarrollo en la realización de pruebas unitarias.
- Describir herramientas utilizadas y técnicas aplicadas para las pruebas unitarias en la plataforma J2EE.
- Entrenar en la aplicación de las buenas prácticas a utilizar en la realización de pruebas unitarias de software y en las técnicas identificadas en la tarea anterior.
- Implementar pruebas unitarias en un caso de estudio que muestren el uso de técnicas y herramientas en la plataforma J2EE.

El contenido del trabajo está seccionado por capítulos de la siguiente manera:

- En el primer capítulo se abordan temas relacionados con las pruebas automatizadas. Se exponen sus principios, características y propósitos, así como estrategias que se suelen aplicar para realizarlas. Se particulariza en las pruebas unitarias de software, mencionado aspectos propios de este tipo de pruebas. Se proponen buenas prácticas a ser aplicadas en proyectos específicamente construidos en la plataforma J2EE.
- En el segundo capítulo se introduce al framework JUnit como herramienta de la plataforma J2EE, así como los elementos que este brinda para realizar las pruebas automatizadas. Se explica cómo escribir pruebas unitarias basadas en JUnit, describiendo y aplicando los

mecanismos que puedan ser utilizados para probar distintos tipos de componentes, así como otras herramientas que pueden ser utilizadas en tales propósitos.

- En el tercer capítulo, se muestra un caso de estudio representando un sistema a desarrollarse en la plataforma J2EE. En el mismo se exponen herramientas con tecnología de punta que brindan soporte para integrar pruebas basadas en JUnit. Se muestra la implementación de pruebas unitarias a distintos tipos de componentes, usando diferentes mecanismos que puedan servir posteriormente como material de consulta.

CAPÍTULO 1: FUNDAMENTOS DE LAS PRUEBAS AUTOMATIZADAS DE SOFTWARE.

1.1 Introducción.

Según la IEEE, se define como pruebas de sistema al proceso de analizar elementos de software para detectar las diferencias entre las condiciones existentes y las condiciones requeridas (errores) y evaluar así las características de dichos elementos (IEEE 1998). En el caso de las pruebas automatizadas (o automáticas) de software son fundamentalmente aquellas que no necesitan de intervención manual para su ejecución. Estas son llevadas a cabo por herramientas que permitan la ejecución de las operaciones del sistema bajo condiciones controladas y evaluación de los resultados.

En este capítulo se presentan las pruebas automatizadas de software (a partir de ahora, se mencionará “pruebas automatizadas” o simplemente “pruebas”), los principios que las regulan y los objetivos de su realización, destacando su importancia, sus ventajas y desventajas dentro del proceso de desarrollo. Se describen diferentes criterios y estrategias que suelen ser usados en su realización tanto de manera general como en la plataforma J2EE.

1.2 Pruebas automatizadas de software.

1.2.1 Propósitos de las pruebas.

Es importante conocer por qué se automatizan las pruebas de software. Entender los propósitos generales posibilitará al desarrollador identificar en qué pueden serles útiles en cada momento. En muchos casos, se tiene la idea de que las pruebas son “algo bueno”, sólo que a veces no se está consciente del porqué. A esta idea se pueden aplicar los criterios siguientes:

- Las pruebas deben ayudar a mejorar la calidad.
- Las pruebas deben ayudar a entender el Sistema Bajo Prueba[1] (SBP)
- Las pruebas deben ayudar a reducir (y no introducir) riesgos.
- Las pruebas deben ser fáciles de ejecutar.
- Las pruebas deben ser fáciles de escribir y mantener
- Las pruebas deben requerir el mínimo mantenimiento mientras evoluciona el sistema.

[1] Es la parte del sistema o el sistema completo en sí que es ejecutado para probar su funcionalidad.

Los tres primeros indican cuál es el valor que aportan las pruebas, mientras que los últimos tres están más relacionados con las características de las pruebas en sí. Estos, a su vez, pueden quedar desglosados en otros más específicos. A partir de este momento se explica cada principio y se identifica en las pruebas lo que se lograría estando presente el principio en cuestión.

Las pruebas deben ayudar a mejorar la calidad.

Gerard Meszarros dice que la razón tradicional por la cual hacer las pruebas es la de asegurar la calidad[2]. ¿Qué se entiende precisamente por esto? ¿Qué es calidad? Definiciones tradicionales diferencian dos categorías principales de calidad, basadas en las siguientes preguntas: (1) ¿Se construye el software correctamente? (2) ¿Se tiene el software correcto?"(MESZAROS 2007)

Las pruebas deben tributar a que los desarrolladores comprueben si el software que construyen es eficaz y eficiente.

- **Las pruebas como especificación** - cuando el proceso de desarrollo es guiado por las pruebas[3] o tan solo se escriben las pruebas antes de desarrollar[4], las pruebas brindan una manera de capturar lo que el SBP debe hacer antes de que se comience a implementar. Se obtiene una vía de especificar el comportamiento en varios escenarios, capturados en una forma que puedan ser ejecutados. Pensar a través de varios escenarios con suficientes detalles en las pruebas ayudará a identificar aquellas áreas donde los requerimientos son ambiguos o contradictorios por sí mismos. Mejora la calidad de la especificación con la cual se mejora la calidad del software que se especifica.
- **Las pruebas detectan errores** - una de las principales funciones de las pruebas es detectar errores en el sistema. Pero, también puede pensarse en las pruebas como "repelentes de errores". No se tendrán errores si primero se ejecutan pruebas de regresión, porque estas señalarán el punto donde se debe verificar el error introducido en el código.
- **Localización de errores** - en ocasiones es más costoso prevenir errores que arreglarlos. La existencia de estos puede ser inevitable. Si se hacen muchas pruebas unitarias[5] medianamente pequeñas que verifiquen un simple comportamiento cada una, se podría encontrar el error rápidamente, conociendo cual es la prueba que falla. Esta es una gran ventaja sobre las

---

[2] Este término se conoce como QA, siglas en inglés de Quality Assurance.

[3] Desarrollo guiado por las pruebas (TDD, siglas en ingles de Test-Driven Development).

[4] Escribir pruebas antes de desarrollar (TFD, siglas en ingles de Test-First Development).

[5] Pruebas realizadas a las partes más atómicas del sistema. Estas pueden ser a clases, métodos, procedimientos, etcétera.

pruebas cliente[6], donde solo se indicaría que algún comportamiento esperado no está funcionando correctamente. Las pruebas unitarias por su parte indicarían claramente el porqué.

Las pruebas deben ayudar a entender el Sistema Bajo Prueba.

Las pruebas pueden brindarle al lector de pruebas[7] un supuesto de cómo es que el código debe funcionar. Las pruebas de caja negra a componentes son un ejemplo de cómo estas describen los requerimientos del componente de software.

- **Las pruebas como documentación** –no es necesario introducirse en el código del SBP para responder cuál debe ser el resultado para una entrada determinada. Las pruebas, por sí solas, dicen cual debe ser el resultado esperado. Las pruebas auto-verificadoras indican la salida esperada a través de una o más aserciones.

Las pruebas deben ayudar a reducir (y no introducir) riesgos.

Las pruebas deben mejorar la calidad del software a través de una mejor documentación de los requerimientos y detectando errores con lo cual reducen el número de riesgos. Otra forma de reducir riesgos es verificar el comportamiento del software en circunstancias "imposibles" que no pueden ser inducidas en las *pruebas cliente* a la aplicación entera como una caja negra. Es importante revisar todos los riesgos del proyecto y determinar cuáles pueden ser mitigados, al menos parcialmente, a través de pruebas completamente automatizadas.

- **Las pruebas como una red de seguridad** – las pruebas actúan como una "red de seguridad" que permite hacer cambios sin miedo a cometer errores. La efectividad de la red de seguridad está determinada por cómo las pruebas verifiquen completamente el comportamiento del sistema. Si se tiene escasez de pruebas se tendrán hoyos en la red de seguridad. Cuando se trabaja con código legado no se tiene una suite de pruebas de regresión. Hacer cambios a este código es riesgoso pues no se conoce qué se puede romper ni si se ha roto algo. El trabajo se torna lento y de manera cuidadosa. Sin embargo, cuando se trabaja con código que cuenta con una suite de pruebas de regresión es posible trabajar más rápido. En este caso las pruebas funcionan como la "red de seguridad" que permite acometer cambios.
- **Las pruebas no deben hacer daño** - se debe ser cuidadoso de no introducir nuevos tipos de problemas en el SBP como resultado de ejecutar pruebas automatizadas. Tener algún código

---

6 Pruebas realizadas por el usuario final del producto para validar su aceptación, ya sea una persona u otro software.

7 Persona que revisa visualmente el código o la configuración de la prueba creada.

de prueba dentro del código de producción es un riesgo. A pesar de desear diseñar un sistema que permita ser probado fácilmente, ningún código específico de prueba debe ser insertado a no ser por la prueba y solamente en el ambiente de pruebas. Se debe mantener la lógica de las pruebas fuera del código de producción del SBP. Otro riesgo, es creer que algún código es confiable sólo porque ha sido probado muchas veces, cuando no siempre es así.

Las pruebas deben ser fáciles de ejecutar.

Las pruebas automatizadas brindan una red de seguridad, permitiendo que el trabajo sea más ágil. No obstante, para poder realizar las pruebas frecuentemente es necesario que sean fáciles de ejecutar:

- **Pruebas completamente automatizadas** –deben ser ejecutadas sin intervención manual, lo cual es un prerrequisito a otros muchos aspectos deseados.
- **Pruebas auto-verificadoras** –deben contener en su código cada aspecto que necesita ser verificado.
- **Pruebas repetibles** –deberían poder ser ejecutadas varias veces, obteniendo siempre el mismo resultado.
- **Pruebas independientes** – es ideal que las pruebas puedan ser independientes y que puedan ser ejecutadas por sí mismas. Así se convertirían en pruebas repetibles sin esfuerzo adicional. Cuando comparten características por muy pequeñas que sean, tales como datos persistentes, usualmente no son repetibles y deben encargarse de eliminar rastros que interfieran en el resultado de otras pruebas.

Las pruebas deben ser fáciles de escribir y mantener.

Cuando se escriben las pruebas, los desarrolladores deben enfocarse en lo que se probará más que en el código de la prueba en sí. Con este fin deben procurar que las pruebas sean sencillas de leer y de escribir. Si una prueba falla por la introducción de un error en su código, es necesario que al revisarla “resalte a los ojos” el error y pueda ser corregido rápidamente. Por otro lado, si el comportamiento de una parte del sistema cambia, muchas pruebas serán afectadas, pero estas deben ser un número relativamente pequeño. El solapamiento sobre el mismo código no ayuda a la mejora del mismo si las pruebas hacen exactamente lo mismo.

Realizar pruebas puede ser una labor complicada por dos razones fundamentales: se trata de verificar muchas funcionalidades en una sola prueba; existe una “brecha de expresividad” muy grande entre el lenguaje en el que se escribe la prueba y los conceptos de dominio que se tratan de expresar en la prueba.

- **Pruebas simples** – las pruebas simples pueden lograrse manteniendo pequeñas pruebas que verifiquen solo una condición a la vez. Cuando se escribe código con el fin de pasar una prueba a la vez, se puede ir agregando porciones pequeñas de comportamiento nuevo al SBP después de pasar cada prueba.
- **Pruebas expresivas** – disminuir la brecha de expresividad puede ser enfocada en crear una librería de métodos útiles para las pruebas que constituyan un lenguaje de dominio específico para las pruebas. El probador podrá expresar los conceptos que desea sin necesidad de traducir sus pensamientos en códigos muy detallados. De esta manera se evita también la duplicación de código en las pruebas, lo cual es un principio no solo aplicable al código de producción.
- **Separar preocupaciones** – esto es aplicable en dos aspectos fundamentales: primero, mantener el código de prueba separado del código de producción; y segundo, enfocar cada prueba en una sola cuestión a verificar. Cuando se verifican dos o más cuestiones a la vez, una de estas puede fallar y procurará que la prueba falle cuando las otras cuestiones podrían no haberlo procurado.

Las pruebas deben requerir mínimo mantenimiento mientras evoluciona el sistema.

El cambio es algo normal y ocurre casi todo el tiempo. Las pruebas deben asegurar que los cambios del SBP no las afecten de forma drástica. Requerir mínimos cambios o mantenimiento, mientras el sistema evoluciona, da idea de que las condiciones del sistema se van comportando de manera estable.

- **Pruebas robustas** – inevitablemente, a medida que evoluciona el sistema pueden existir muchos cambios que afecten las pruebas. Se debe tratar que el número de pruebas afectadas sea mínimo. Una de estas maneras es evitando el solapamiento de las pruebas. Otra razón puede ser el cambio de ambiente en las pruebas. En lo posible debe probarse el sistema de manera aislada. Verificar una condición a la vez y utilizar métodos útiles para las pruebas puede ayudar a reducir en mucho el número de pruebas que serán afectadas directamente por algún cambio.

1.2.2 Ventajas y desventajas de las pruebas.

Las pruebas automatizadas permiten a los desarrolladores ser mucho más audaces en el modo en que modifican el software existente. Esto se debe a que permiten una forma más evolutiva de desarrollo soportando una entrega incremental de funcionalidades para el cliente, acelera la retroalimentación por parte del usuario y mejora la calidad de la aplicación en construcción.

Por el gran aporte que representan, a pesar de que en un principio fue una práctica fundamental dentro de métodos de desarrollo ágil tal como XP[8], su uso ha sido difundido también para métodos menos ágiles de desarrollo[9].

En general, las pruebas automatizadas son más repetibles que las manuales. Procuran ser exactamente igual todo el tiempo y no olvidan ningún detalle. Consumen menos esfuerzo para ser ejecutadas que las pruebas manuales, por lo que es posible que sean realizadas más a menudo.

Los desarrolladores pueden ejecutar las pruebas cada vez que modifiquen el código de producción y necesiten obtener un voto de confianza de manera rápida. Conciben las pruebas como una red de seguridad que trata de atrapar errores y les permite trabajar con más rapidez y menos paranoia. El desarrollo se hace más productivo, independientemente del esfuerzo adicional implicado en escribir las pruebas.

Sin embargo, escribir buen código de prueba no es sencillo. Cuando se hace, en muchos casos, existe una fuerte tendencia o tentación a rendirse de seguir escribiéndolo. Es cierto que siempre habrá un costo adicional para crear y mantener una suite de pruebas automatizadas. Muchos desarrolladores experimentados sostienen el criterio de que es peor emplear y gastar más tiempo tratando de encontrar el motivo por el cual algo está fallando.

El esfuerzo y el costo de una prueba automatizada radican en la repetición, el mantenimiento y la comunicación con el SBP. La repetición de resultados requiere repetición de configuración de la estructura de las pruebas y repetición de interacciones con el SBP. El mantenimiento es inevitable a medida que el sistema va evolucionando, solo que el grado de cambio estará dado por la manera en que se diseñen las pruebas. La comunicación será posible a través de interfaces dentro del SBP que permitan establecer el estado correcto antes y encontrar el estado en el que está después de la prueba. Siempre procurando que las pruebas sean fáciles de entender.

Por ello, es importante tener en cuenta varios aspectos cuando se automatizan las pruebas, los cuales pueden determinarse a través de las siguientes preguntas:

- ¿Cómo se interactúa con el SBP?
- ¿Cuál es la mejor manera de expresar el resultado esperado?
- ¿Cómo evitar que una prueba afecte a otra prueba?
- ¿Cómo asegurar que las pruebas tengan el mismo efecto a medida que evoluciona el SBP?

---

[8] XP, siglas en ingles de eXtreme Programming.

[9] Esto es posible a través de la introducción del Desarrollo Dirigido por Pruebas (TDD, siglas en inglés de Test-Driven Development), como un proceso alternativo menos extremo.

- ¿Cómo probar el SBP cuando depende de otro software o sistema que no ha sido escrito aún?
- ¿Puede ser usado en el ambiente creado para las pruebas?
- ¿Toma mucho tiempo ejecutar las pruebas?
- ¿Debemos dejar de hacer las pruebas cuando se vuelvan difíciles de escribir y tenga pocos beneficios?

Hacer buenas soluciones es un reto pero a medida que se vaya obteniendo experiencia en escribir pruebas será más fácil hacerlas.

1.2.3 Filosofías de las pruebas.

Diferentes filosofías de hacer las pruebas automatizadas son basadas en la experiencia acumulada de desarrolladores. Tener varios puntos de vistas permite tener varios criterios acerca de cómo aprovechar las pruebas en cierto momento. Es por ello que algunos pueden utilizar siempre una herramienta mientras otros lo hacen muy poco o, simplemente, las emplean de una manera y no de otra.

Conocer las diferencias entre varias formas de trabajar con las pruebas permitirá determinar con mayor razón el uso de las mismas. Las contradicciones más comunes pueden encontrarse en los siguientes criterios:

- Probar primero o probar después.
- Escribir prueba por prueba o todas las pruebas al mismo tiempo.
- Probar de afuera hacia adentro o de adentro hacia afuera.
- Verificar comportamiento o estado.
- Configuración diseñada prueba por prueba o configuración diseñada para un conjunto de pruebas.

De cada una de estas contradicciones se presenta una valoración de las ganancias y pérdidas de aplicar cada una de las partes que se contraponen.

Probar primero o probar después.

En el desarrollo tradicional de software las pruebas se preparan y ejecutan después de que todo el software es diseñado y codificado. Algunas metodologías ágiles como XP, han promovido el escribir las pruebas primero antes de comenzar a escribir algún código del producto.

En la práctica, cualquier desarrollador podría asegurar que es más difícil escribir una suite de pruebas automatizadas después de haber escrito el código de producción. Incluso, aunque se diseñe el sistema

para que sea fácilmente probado, la probabilidad de que no haya que cambiar algo es baja. Sin embargo, al escribir las pruebas primero, el diseño del sistema será inherentemente fácil de probar.

Cuando se escriben las pruebas primero, el desarrollador se interesa en escribir el código necesario y suficiente que sea capaz de pasarlas. Las funcionalidades opcionales suelen no ser escritas y la producción del código tiende a ser mínimo. Es posible prever detalles que no se tomarán en cuenta en la implementación. No hay esfuerzo en buscar un error por muy simple que este sea. Las pruebas determinarán qué está mal y qué está bien a medida que se vaya trabajando. Además, las pruebas suelen ser más robustas.

Escribir prueba por prueba o todas las pruebas al mismo tiempo.

El proceso de Desarrollo Dirigido por Pruebas (de ahora en lo adelante TDD, siglas en inglés de Test-Driven Development) promueve la idea de escribir una prueba y luego escribir el código que pase la prueba. Al crear una prueba e ir escribiendo el código de manera incremental, se puede estar concentrado en que sólo una prueba falla. En estos casos, no hay necesidad de utilizar depuradores de código, pues la separación de las pruebas y el desarrollo incremental dejan poca duda del por qué falla la prueba.

Este no es el caso para todas las formas de escribir pruebas antes que el código. Otra variante es identificar todas las pruebas necesarias para la condición en cuestión antes de comenzar a codificar. Esto permite pensar como lo haría el cliente o el probador, evitando preocuparse tan temprano por la solución.

Probar de afuera hacia adentro o de adentro hacia afuera.

Cuando se diseña el software desde afuera[10], se piensa en las pruebas clientes de caja negra para el sistema completo y luego en las pruebas unitarias para cada pieza del software que se diseña. Estas pruebas hacen pensar como cliente más que como desarrollador. El enfoque de las pruebas va dirigido a las interfaces que se proveen al usuario del software, ya sea una persona u otra pieza de software. A medida que se avanza se pueden implementar pruebas a los componentes que se decidan construir.

Algunos desarrolladores de pruebas prefieren diseñar y codificar desde afuera hacia adentro. Esto fuerza a tener un problema de dependencia con componentes que no han sido construidos aún. Incluso se pueden utilizar los agentes dobles para proveer de entradas indirectas que son imposibles al SBP –tales como valores fuera de rangos o excepciones-, y verificar que es capaz de manejarlas

[10] Desde una perspectiva en que se ven las funcionalidades del sistema de manera general y luego se detallan las cuestiones específicas del mismo.

correctamente. Una vez que los componentes de los cuales se depende estén construidos, se pueden ir eliminando los agentes dobles de muchas pruebas. El mantenerlos brinda una mejor localización de errores, pero aumenta el costo de mantenimiento de las pruebas. En la Figura 1.1 se observa cómo se suele utilizar agentes dobles[11], para que la capa más externa pueda ser ejecutada y probada.

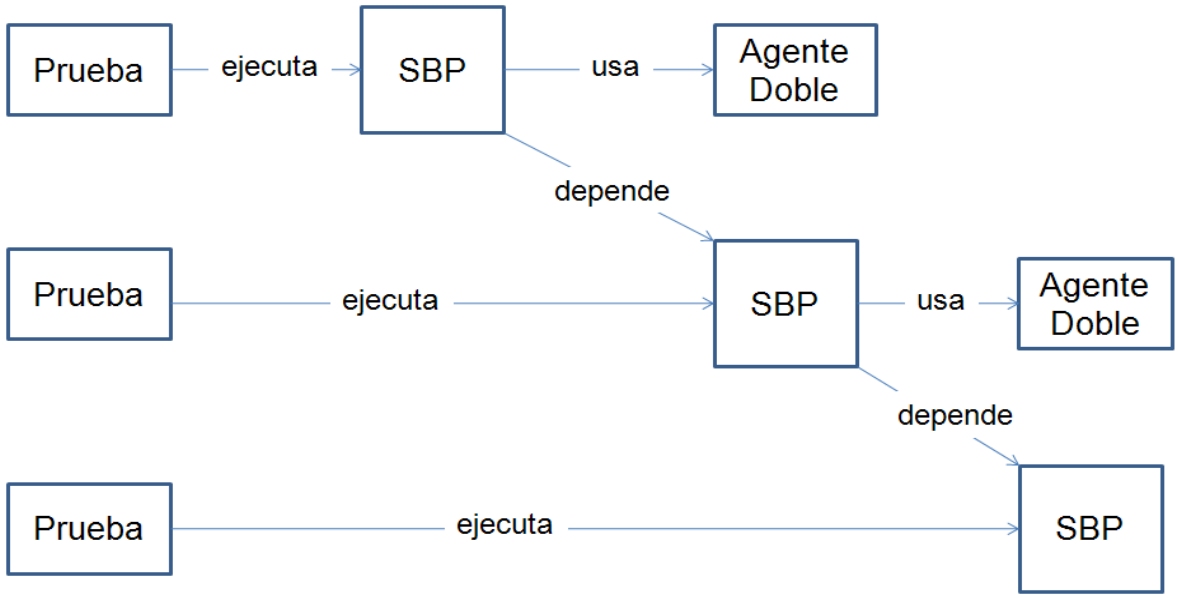

Figura 1.1 Desarrollo de afuera hacia adentro. Las pruebas al SBP usan agentes dobles en lugar de otras piezas de software de las que depende.

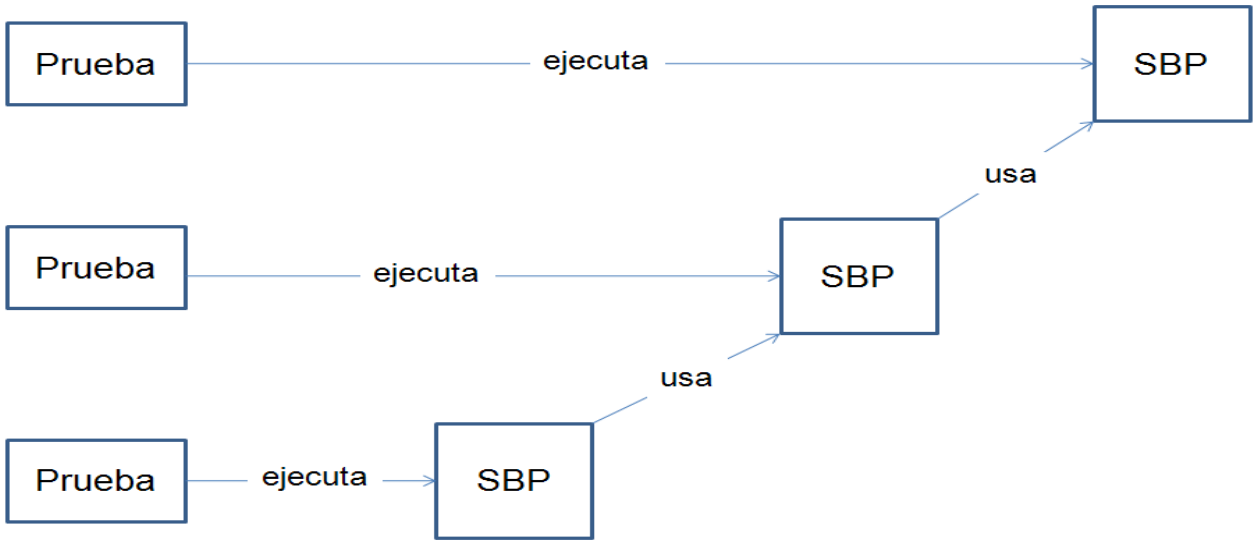

Figura 1.2 Desarrollo de adentro hacia afuera a partir de los componentes internos hacia los más externos.

[11] Elementos que simulen o sustituyan parcialmente la funcionalidad de un componente para hacer posible la ejecución de la prueba

Sin embargo, otros desarrolladores prefieren diseñar y luego codificar de adentro hacia afuera[12], con el fin de evitar un problema de dependencia con componentes que no estén escritos. Esto requiere anticipar la necesidad del software externo cuando se escriben las pruebas del software interno. No se prueba el software externo separado del software interno, por lo que se evita utilizar agentes dobles en muchas pruebas. Por otro lado, construir de adentro hacia afuera permite separar por capas el SBP y usar esporádicamente los agentes dobles para aquellas entradas indirectas que las capas internas del software no pueden retornar durante el uso normal[13]. Todo esto se muestra en la Figura 1.2.

Verificar comportamiento o estado.

Escribir el código de afuera hacia adentro es un primer paso para verificar el comportamiento en vez del estado. La "vista estática" que brinda es suficiente para poner el SBP en un estado específico, ejecutarlo y verificar que al final de la prueba está en el estado esperado.

La "vista de comportamiento" dice que se debe especificar no solo el estado inicial y final del SBP, sino también las llamadas que este hace a sus dependencias. En estos casos, deben especificarse los detalles de las llamadas a las interfaces de las salidas del SBP[14] que no siempre regresan directamente al cliente o a la prueba.

Al probar el comportamiento[15] se utiliza con frecuencia los agentes dobles en las pruebas puesto que aíslan el SBP de los componentes de los que depende pero incrementan el costo para el mantenimiento y la refactorización de las pruebas.

Configuración diseñada prueba por prueba o configuración diseñada para un conjunto de pruebas.

Para las pruebas se suele tener una aplicación y una base de datos con una variedad de datos de prueba previamente concebidos. El contenido de la base de datos es cuidadosamente diseñado para permitir la ejecución de diferentes escenarios. Cuando la configuración para las pruebas es aprovechada de igual manera, el probador puede definir una configuración estándar que puede ser utilizada por todas las pruebas.

---

[12] Primero se determinan las funcionalidades más atómicas posibles y se orquestan para formar funcionalidades más generales.

[13] Escenarios de errores donde se retornan códigos de errores o se lanzan excepciones.

[14] Estas salidas pueden ser tales como valores retornados por funciones.

[15] Esta manera de escribir las pruebas es también llamado Desarrollo dirigido por comportamiento(BDD, siglas en inglés de Behavior-Driven Development)

La configuración puede ser establecida en cada método de prueba, ya sea a través de una configuración personalizada, o de una configuración delegada en otro método que todos utilicen. Alternativamente, esta configuración puede estar compartida y ser reutilizada por muchas pruebas. De cualquier manera, la dificultad radica en saber qué partes de una configuración son pre-condiciones para un método de prueba en particular.

Por otro lado, el diseño personalizado de configuraciones mínimas para cada prueba puede convertirse en un aprovechamiento más ágil. No existiría una configuración completa para todas las pruebas en su conjunto y se lograría una estrategia más consistente haciendo nuevas configuraciones en cada prueba.

Una estrategia propuesta.

Gerard Meszaros, a criterio personal, propone una manera de trabajo: escribir las pruebas antes que el código en producción; una por una de manera usual, aunque podría crearse una lista de pruebas como especie de esqueleto para comenzar; desarrollar de afuera hacia adentro para determinar lo que se necesita de la capa inmediata inferior; verificar el estado y si, además, se necesita tener en cuenta la cobertura del código en las pruebas entonces verificar el comportamiento; y realizar diseño de las configuraciones de las pruebas, una por una.(MESZAROS 2007)

1.2.4 Principios de las pruebas.

Anteriormente se conocieron los propósitos para los cuales las pruebas son concebidas dentro del desarrollo. Se mostró también las formas en que los desarrolladores pueden concebir las pruebas automatizadas. Los principios indican el porqué se debe proceder de la manera adecuada y las consecuencias de hacerlo o no. Gerard Mezsaros plantea que vale la pena esforzarse en seguir los siguientes principios, y desviarse de ellos sólo cuando se esté completamente consciente de las consecuencias (MESZAROS 2007).

- Escribir primero las pruebas.
- Diseñar el sistema para que sea fácil de probar.
- Interactuar con interfaces como fachada del SBP.
- Comunicar la intención de la prueba.
- No modificar el SBP.
- Mantener independencia entre las pruebas.
- Aislar el SBP.
- Minimizar el solapamiento de las pruebas.

- Minimizar el código difícil de probar.
- Mantener la lógica de las pruebas fuera del código de producción.
- Verificar una condición por prueba.
- Probar funcionalidades de manera separada.
- Asegurar el equilibrio entre esfuerzo y responsabilidad.

Para no contradecir estos criterios durante el desarrollo del sistema, el probador debe velar y enfocarse en algunos aspectos que han de ser evitados. Estos aspectos, en el momento en que aparezcan pueden ser síntomas de problemas que existen en las pruebas.

Escribir primero las pruebas.

El Desarrollo Dirigido por Pruebas (de ahora en lo adelante TDD según sus siglas en inglés: Test-Driven Development) viene dado por una práctica que se va adquiriendo. Una vez que el desarrollador haya empleado esta manera de trabajo le parecerá extraño hacerlo de otra manera, como mismo le es extraño el TDD a quien nunca lo haya hecho. De este modo, las pruebas automatizadas evitan el esfuerzo de depuración, que a menudo contrarresta el costo de automatizarlas. Escribirlas antes del código fuerza a que ocurra el principio que se explica a continuación.

Diseñar para que el sistema sea fácil de ser probado.

Esto podría parecer redundante a lo anterior pero no es así. En el caso que se ignore el principio anterior, se hace más necesario e importante el realizar un diseño que permita ser probado fácilmente. En el caso de los sistemas legados, es difícil que puedan ser probados si previamente a su diseño no se tomaron en cuenta estas consideraciones.

Interactuar con interfaces como fachada del SBP.

Los objetos tienen varios tipos de interfaces, tanto públicas como privadas, que pueden ser utilizadas por clientes o clases más cercanas, incluso algunos pueden tener interfaces de salida que sean utilizadas por parte de muchos objetos de los cuales dependen.

Las interfaces que utilizan las pruebas influyen en la robustez de las mismas. El uso de estas, para la manipulación de “puertas traseras” en las configuraciones del SBP o en la verificación de las salidas esperadas, puede ser útil cuando el software es altamente acoplado, donde es más frecuente el mantenimiento de las pruebas.

El uso de verificación de comportamiento y los agentes dobles puede no ser factible en el software que sea sobredetallado[16], donde las pruebas tienden a ser más inestables debido a continuos cambios, provocando que los desarrolladores desistan de refactorizaciones deseables.

Cuando todas las opciones son igualmente efectivas, se pueden hacer "pruebas de ida y vuelta", donde se prueba un objeto a través de sus interfaces públicas y se verifica el estado para determinar si se ha comportado correctamente. Si esto no es suficiente para describir el comportamiento esperado se pueden hacer "pruebas cruzadas", donde se verifica el comportamiento y se determina si el SBP hace las llamadas a los componentes de los que depende. De ser necesario se pudieran emplear agentes dobles.

Comunicar la intención de la prueba[17].

Las pruebas automatizadas son programas. Necesitan estar correctas sintácticamente para compilar y semánticamente para ser ejecutadas de manera exitosa. Necesitan implementar cualquier lógica detallada que sea necesaria para poner el SBP en un estado inicial apropiado y verificar que la salida sea la esperada. Estas cuestiones son necesarias pero no son suficientes.

Puede que se descuide el hecho de que las pruebas posteriormente necesitarán ser leídas y entendidas para efectuar el mantenimiento. Las pruebas que contienen mucho código[18] son usualmente pruebas oscuras puesto que son difíciles de entender y/o de saber qué es lo que prueban exactamente con solo mirar el código. Para comprenderlas es imprescindible tener una vista completa y considerar todos los detalles. Esto toma tiempo extra cada vez que es necesario revisar la prueba para darle mantenimiento o usarla como documentación.

Las pruebas deben ser legibles y fáciles de mantener, es decir, dar a conocer claramente el objetivo de la prueba en cuestión. Una manera de lograrlo es haciendo llamadas a métodos útiles con nombres descriptivos de sus objetivos, ya sea para configurar la prueba o para verificar que la salida es la esperada.

Debería ser fácilmente visible cómo la configuración influye en la salida de cada prueba. Una surtida librería de métodos útiles también hace que las pruebas sean más fáciles de escribir y entender, pues no se necesita codificar los detalles de la lógica en cada prueba.

---

[16] Es el software que requiere la implementación de muchas especificaciones que evitan su abstracción en casos más generales.

[17] En la bibliografía consultada este principio se encuentra con la terminología lenguaje de alto nivel o legible a simple vista.

[18] Consideradas cuando posee un número mayor de 10 líneas o mucha lógica condicional.

No modificar el SBP.

A menudo, algunas pruebas requieren reemplazar partes de la aplicación con agentes dobles. Esto se puede hacer cuando se necesite el control de las entradas indirectas o cuando se necesite hacer verificación en el comportamiento del SBP interceptando sus salidas indirectas, incluso cuando algunas partes tienen efectos colaterales no deseables o dependencias que son imposibles de satisfacer durante el desarrollo o el ambiente de prueba.

Al modificar el SBP se corre el riesgo de que ya no se esté probando el código que se planea poner en producción. Debe asegurarse que se está probando el software en una situación que sea representativa de la manera en que se usará en producción, sino se estaría reemplazando la parte del SBP que se pensaba probar.

Mantener independencia entre las pruebas.

Cuando se realizan pruebas manuales es común que varias de estas compartan la misma configuración y por tanto se verifiquen muchos aspectos del comportamiento del SBP en una sola ejecución. La consecuencia de esta unión es simplificar el esfuerzo minimizando el número de pasos. En estos casos, los probadores tienen que tener la habilidad de reconocer cuándo el fallo de una prueba pueda imposibilitar la continuidad de la misma, cuándo causará que otras pruebas sean descartadas y cuándo será irrelevante en el resultados de otras.

En el caso de las pruebas automatizadas, si las pruebas son interdependientes o, peor aún, son orden-dependientes, el probador no podrá aprovechar toda la retroalimentación que las pruebas podrían aportar. Las pruebas interactivas tienden a fallar en grupo, pues la falla en una prueba provocaría inevitablemente la falla de las pruebas dependientes. Cuando dos pruebas fallan, no habría manera de decir si es por problema de ambas o es sólo problema de la primera; más difícil sería si ocurriera lo mismo con muchas pruebas a la vez.

Una prueba independiente es la que puede ser ejecutada por sí sola. Su configuración coloca al SBP en un estado que pueda verificar su comportamiento. Las fallas, sin duda alguna, identificaría la localización del error necesario para ayudar a marcar el lugar del código donde se encuentra.

Aislar el SBP.

La mayoría de las piezas del software se construyen a partir de otras piezas de software desarrolladas por terceros, por ejemplo un componente de una librería, o dentro de la misma aplicación. Cuando se depende de un software que puede cambiar con el tiempo, las pruebas pueden comenzar a fallar

repentinamente porque el comportamiento del software ha cambiado inevitablemente[19]. Cuando el software a probar depende de otro cuyo comportamiento no se puede controlar, puede ser difícil verificar que se comporta apropiadamente con todos los posibles valores obtenidos. Es como lidiar con código que no ha sido probado.

Cualquier aplicación, componente, clase o método que se esté probando, debe aislarse tanto como sea posible de todas las otras partes de software que no se hayan elegido probar. Las pruebas podrán comportarse de manera robusta, reduciendo la sensibilidad de contexto causada por el acoplamiento entre el SBP y el software que lo rodea.

Esto puede ser favorecido si se diseña de manera tal que cada pieza de software de la que se dependa pueda ser reemplazada por un agente doble. Así se puede tomar el control de las entradas indirectas del SBP y tener pruebas más repetibles y robustas.

Minimizar el solapamiento de las pruebas.

La mayoría de las aplicaciones tienen muchas funcionalidades a verificar. Probar que todas trabajan correctamente en todas las combinaciones y todos los escenarios es casi imposible. Escoger las pruebas a escribir debe ser una actividad priorizada dentro de la gestión de riesgos.

Las pruebas se deben estructurar de modo que existan pocas que prueben una misma pieza o funcionalidad del software. Las pruebas que verifican el mismo código frecuentemente fallan al mismo tiempo y tienden a necesitar igual mantenimiento cuando el SBP es modificado.

Se debería asegurar, en lo posible, que todas las condiciones de pruebas estén cubiertas por todas las pruebas que se tienen y que cada condición lo sea por exactamente una prueba. Si un objetivo se puede probar de muchas maneras distintas, se deberían identificar diferentes condiciones de pruebas.

Minimizar el código difícil de probar.

Muchos tipos de código son difíciles de probar con pruebas completamente automatizadas, por ejemplo: componentes GUI, código multihilo y los mismos métodos de prueba. El problema radica en que generalmente están embebidos en un contexto que los hace difíciles de instanciar o de interactuar con ellos. No cuentan con pruebas automatizadas que lo protejan de errores por lo que se hace difícil modificar, de manera segura, su estructura interna y es más peligroso modificar cuando se deseen introducir nuevas funcionalidades.

[19] Este tipo de dependencia suele llamarse “sensibilidad de contexto”.

Es deseable minimizar la cantidad de código que sea difícil de ser probado y que deba ser mantenido en algún momento. En algunos casos, se puede mejorar esta dificultad moviendo la lógica que se desea probar fuera de la clase o del método que la causa, incluso los métodos de prueba pueden tener más de su código en métodos útiles de prueba que puedan ser probados. Reducirlos mejora la cobertura del código en las pruebas, la confianza en el código y la habilidad de modificarlo.

Mantener la lógica de las pruebas fuera del código de producción.

En ocasiones, cuando el código de producción no es diseñado para ser fácil de probar, se escriben sentencias que toman forma de prueba y que ejecutan acciones alternativas o simplemente previenen que se ejecuten otras. Estas lógicas embebidas en el código que actúan cuando se prueba el software podrían causar fallas en la producción.

Las pruebas están pensadas para verificar el comportamiento del sistema. Si el sistema se comporta diferente cuando se encuentra bajo pruebas entonces es difícil ser certero en afirmar que el código de producción realmente funciona.

“Un sistema bien diseñado, desde la perspectiva de las pruebas, es aquel que prevé el aislamiento de funcionalidades. Los sistemas orientados a objetos son más propicios a ser probados porque están compuestos de objetos discretos pero, desafortunadamente, incluso ellos pueden ser construidos de manera que sean difíciles de probarse y se puede encontrar embebido código con lógica de pruebas” (MESZAROS 2007).

Verificar una condición por prueba.

Muchas pruebas requieren que el SBP se encuentre en un estado inicial particular mientras otras solo demandan un estado por defecto. Como el sistema transita por varios estados desde que comienza a probarse hasta que se finaliza, puede existir una fuerte tentación a reutilizar el estado final de una condición de prueba como estado inicial de otra.

Uno de los inconvenientes es que si una aserción falla, el resto de la prueba no es ejecutada, por tanto, no se obtendrá información sobre qué parte del sistema trabaja y cuál no. Se hace más difícil la localización de errores.

Cada prueba pensada debe verificar una sola condición. Para ello se sugiere establecer configuraciones para cada una, donde no habría problema si se ejecuta la misma secuencia de pasos varias veces. En caso de que necesitaran la misma configuración, se pueden mover los métodos de prueba a una misma clase de prueba y establecer una configuración común para todos.

Un posible aspecto que argumenta la razón de verificar una condición por prueba es qué se entiende por “una condición”, según el contexto en el que se encuentre. Algunos probadores pueden entender

que debe ser una aserción por prueba y por consiguiente una por cada método de prueba. Esto hace que se deban escribir más métodos de pruebas. Sin embargo, otros determinan hacer “aserciones personalizadas” o “métodos de verificación”. En estos casos se hacen llamadas a métodos que por su nombre pueden indicar el objetivo a verificar. Se reducen las múltiples aserciones y la prueba se torna más legible.

Probar funcionalidades de manera separada.

El comportamiento de una aplicación compleja se basa en la agregación de muchos comportamientos más pequeños. En ocasiones estos comportamientos son suministrados por un mismo componente. Cada uno de estos comportamientos representa una funcionalidad diferente y puede tener un número significativo de escenarios en los cuales necesite ser verificado.

Probar muchas responsabilidades en una misma prueba puede implicar que toda la prueba falle cuando alguna de estas falle. Sería peor cuando no se pueda identificar cuál es la que implica la falla. Como no es posible tener una buena localización de errores usualmente requiere depuración manual y tomará más tiempo en detectar y corregir el problema. Otro inconveniente podría ser el hecho de que al probar una funcionalidad nunca se pase por un estado en el que podría fallar puesto que pruebas precedentes a otras funcionalidades no colocan al SBP en el estado requerido.

Probar separado permite tener el control de poder decir en qué parte específica del sistema se encuentra la falla y hace más fácil de entender el comportamiento que será probado.

Asegurar el equilibrio entre esfuerzo y responsabilidad.

El esfuerzo que toma escribir pruebas o modificarlas no debería exceder al esfuerzo que toma implementar la correspondiente funcionalidad. Asimismo, las herramientas requeridas para las pruebas no deben requerir más experticia que las herramientas utilizadas en implementar las funcionalidades a probar.

1.2.5 Estrategias en las pruebas.

Una decisión estratégica en el caso de las pruebas, sería una decisión que cambiarla afectaría un gran número de pruebas; especialmente donde muchas o todas pueden ser convertidas en muchas estrategias de prueba. Es muy costoso el esfuerzo del cambio, por lo que siempre se hace necesario tener algunas consideraciones en cuenta:

- ¿Qué tipos de pruebas se automatizan?
- ¿Qué herramientas usar para automatizar las pruebas?

- ¿Cómo administrar la configuración de las pruebas?
- ¿Cómo asegurar que el sistema sea fácil de probar?
- ¿Cómo hacer interactuar las pruebas con el SBP?

Cada decisión estratégica puede tener consecuencias de gran alcance. Es bueno hacerlas conscientemente en el momento correcto basadas en la información disponible.

¿Qué tipos de pruebas se automatizan?

Existen muchos puntos de vistas por las cuales se puede o no definir tipos de pruebas. Uno es que divide las pruebas en las siguientes categorías:

- **Pruebas de funcionalidades cruzadas** – verifican varios aspectos del comportamiento del sistema que interrelacionan funcionalidades específicas. Generalmente son manuales y se realizan para criticar y valorar el producto.
- **Pruebas por funcionalidad** – verifican el comportamiento del SBP en respuesta a un estímulo en particular. Prueban el sistema a varios niveles. Estas pruebas son usualmente automatizadas y forman parte del apoyo al desarrollo del producto.

En el caso de las pruebas por funcionalidad[20] verifican el comportamiento directamente observable de una pieza de software en respuesta a un estímulo específico. La funcionalidad puede estar relacionada con el negocio o con requerimientos operacionales, incluyendo el mantenimiento del sistema y escenarios específicos de tolerancia a fallas. Estos requerimientos pueden ser expresados como casos de uso, historias de usuario o escenarios de prueba, etcétera.

En consecuencia, se pueden subdividir según los propósitos que persiguen en: pruebas de aceptación, pruebas de componentes y pruebas unitarias. En la Figura 1.3 se muestra la clasificación de estas pruebas según sus propósitos.

- **Pruebas de aceptación** - verifican el comportamiento completo del sistema o de la aplicación en cuestión. Típicamente corresponden a escenarios de uno o más casos de uso, características o historias de usuario. Pueden ser automatizadas por desarrolladores, pero la característica clave es que el usuario final debe ser capaz de reconocer el comportamiento especificado por la prueba, incluso si estos no pueden leer la representación de la prueba.
- **Pruebas unitarias** - verifican el comportamiento de sólo una clase o método, que es consecuencia de una decisión de diseño. Este comportamiento no es típicamente relacionado a

[20] No se abordarán las pruebas de funcionalidades cruzadas por ser pruebas manuales y estar fuera del contexto del tema presentado en el trabajo.

los requerimientos excepto cuando el segmento clave de la lógica de negocio es encapsulado dentro de la clase o del método en cuestión. Son escritas por desarrolladores para su propio uso. Ayudan a describir qué es lo que “parece estar terminado”, resumiendo el comportamiento de la unidad en forma de pruebas.

- **Pruebas de componentes** - verifican componentes que consisten en grupos de clases que colectivamente proveen algún servicio. Estas se encuentran entre las pruebas unitarias y las pruebas del cliente en cuanto al tamaño del SBP que está siendo verificado.

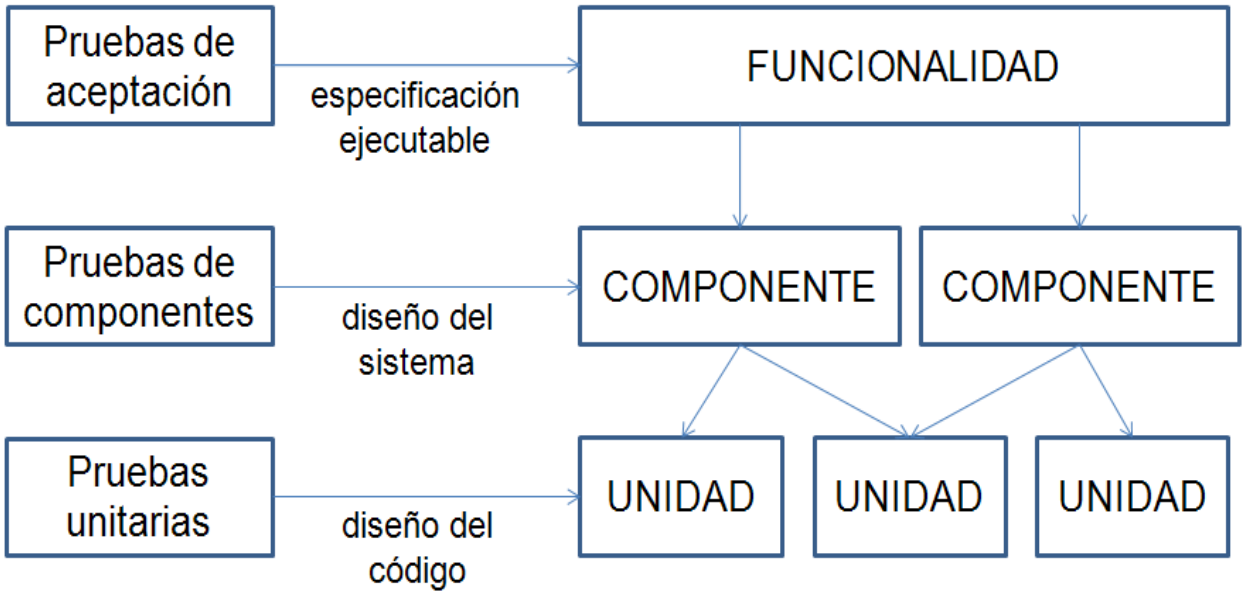

Figura 1.3 Tipos de pruebas automatizadas clasificadas según sus propósitos.

¿Qué herramientas usar para automatizar las pruebas?

Usar las herramientas correctas para el trabajo es tan importante como tener buenas habilidades con ellas. La selección de la adecuada es también una decisión estratégica. Una vez que se haya invertido esfuerzo en el aprendizaje del uso de una herramienta, a medida que se automaticen las pruebas será mucho más difícil cambiarla por otra distinta.

Fundamentalmente, hay dos formas diferentes de automatizar pruebas: una, a través de las pruebas “grabadas o registradas”; otra, a través de las pruebas “escritas a mano”.

- **Pruebas grabadas o registradas** – consisten en usar herramientas que monitorean la interacción con el SBP mientras se hacen pruebas manuales. La información que obtienen es guardada en un fichero o base de datos. Estos datos se convierten en un escenario para reproducir esta prueba nuevamente con otra versión (o quizás la misma) del SBP. El problema principal de estas pruebas es el nivel de granularidad que graban. La mayoría de las

herramientas comerciales graban las acciones a nivel de interfaz de usuario, siendo útiles en las pruebas que no son robustas.

- **Pruebas escritas a mano** – consisten en codificar manualmente los programas que ejecutarán y probarán el sistema. Generalmente se emplean frameworks de pruebas automatizadas, ficheros batch, macrolenguajes y herramientas comerciales o de código abierto.

¿Cómo administrar la configuración de las pruebas?

Para cada prueba se pueden determinar cuatro fases distintas para su realización y su ejecución: establecer configuración, ejecución del SBP, verificación de resultados y desmontar configuración[21]. En la primera de estas fases es donde se establece el estado requerido en el que el SBP se ejecutará y será probado, por lo que toda la prueba depende de ello.

La administración de la configuración de las pruebas tiene gran impacto en la ejecución y en la robustez de las mismas. Seleccionar una mala estrategia no se detecta inmediatamente. Desafortunadamente, cuando se detectan los problemas y la necesidad de cambios se torne visible, el costo será significativo por el gran número de pruebas que puedan estar involucradas.

A partir de los tipos de pruebas que se necesiten realizar, se pueden señalar tres principales estrategias para la configuración de las mismas:

- La más simple sólo requiere que se organice el código para crear la configuración por cada prueba. Colocar el código en los métodos de pruebas y no tomar en cuenta los métodos útiles de prueba. Los elementos configurados se mantienen en memoria solo lo necesario para cada prueba y convenientemente desaparecen tan pronto se haya terminado con ellos.
- La segunda estrategia involucra el uso de configuraciones que por una razón u otra transcienden más allá de un simple método de prueba que las usa. Para evitar que se convierta en una configuración compartida, se requiere de código que explícitamente la destruya al final de cada prueba.
- La tercera estrategia es el utilizar configuraciones persistentes en una fuente de datos común. Estas configuraciones compartidas son usadas a menudo para mejorar la velocidad de ejecución de las pruebas que contienen gran cantidad de carga. Se debe tener cuidado en

[21] En la primera fase, se establece el estado del SBP que es requerido por el inicio de la prueba. En la segunda fase, se ejecuta el SBP y se interactúa con las funcionalidades a probar. En la tercera fase, se verifica si la salida obtenida es la esperada. En la última fase, se debe restablecer el SBP al estado en que se encontraba antes de comenzar la prueba para que no afecte a otras pruebas.

utilizarlas de manera deliberada en muchas pruebas, pues el número de pruebas y la cantidad voluminosa de datos pueden hacerlas incontrolables o difíciles de mantener.

¿Cómo asegurar que el sistema sea fácil de probar?

También es importante asegurar que el sistema permita ser fácilmente probado. Al diseñarlo se debe tener en cuenta algunos criterios:

- **Crear las pruebas de último es un riesgo** – esta manera de crear las pruebas automatizadas las hace un tanto difícil. La mayoría de los beneficios se perciben durante la fase de depuración, donde se reduce considerablemente el tiempo empleado con herramientas de depuración.
- **La facilidad para probar viene dada por las pruebas** – cuando el desarrollo del software va dirigido por las pruebas, no es necesario preguntarse si el sistema será fácil de probar. Escribir pruebas permite definir los puntos de control y los puntos de observación que el SBP debe proveer.
- **Diseñar para el sistema sea fácil probar** – esto puede ser difícil, por lo mismo que es difícil conocer lo que las pruebas podrían necesitar en forma de puntos de control y puntos de observación en el SBP. Es más fácil crear un sistema difícil de probar. De igual manera, es posible que se pierda tiempo en el diseño de mecanismos del sistema que sean insuficientes o innecesarios para que esté pueda ser probado.
- **Crear puntos de control y puntos de observación** – una prueba interactúa con el software a través de una o más interfaces determinando puntos de interacción (estos puntos pueden ser de control o de observación). A través de un punto de control la prueba le pide al software que haga algo por ella y puede ser usado para poner el software en un estado específico como parte de la configuración para ejecutar el SBP. Un punto de observación es para encontrar el comportamiento del SBP durante la fase de verificación de resultados en las pruebas. Con estos puntos de interacción se obtiene el estado posterior del SBP y se espían las interacciones entre le SBP y cualquier componente con el cual se espera que interactúe mientras es ejecutado.

En la Figura 1.4 se muestran cómo son establecidos estos puntos de interacción, donde los puntos de interacción directa son llamadas realizadas por las pruebas a métodos síncronos, los puntos de interacción indirectos requieren alguna manipulación por puertas traseras y los puntos de control apuntan hacia el SBP mientras que los puntos de observación señalan desde él.

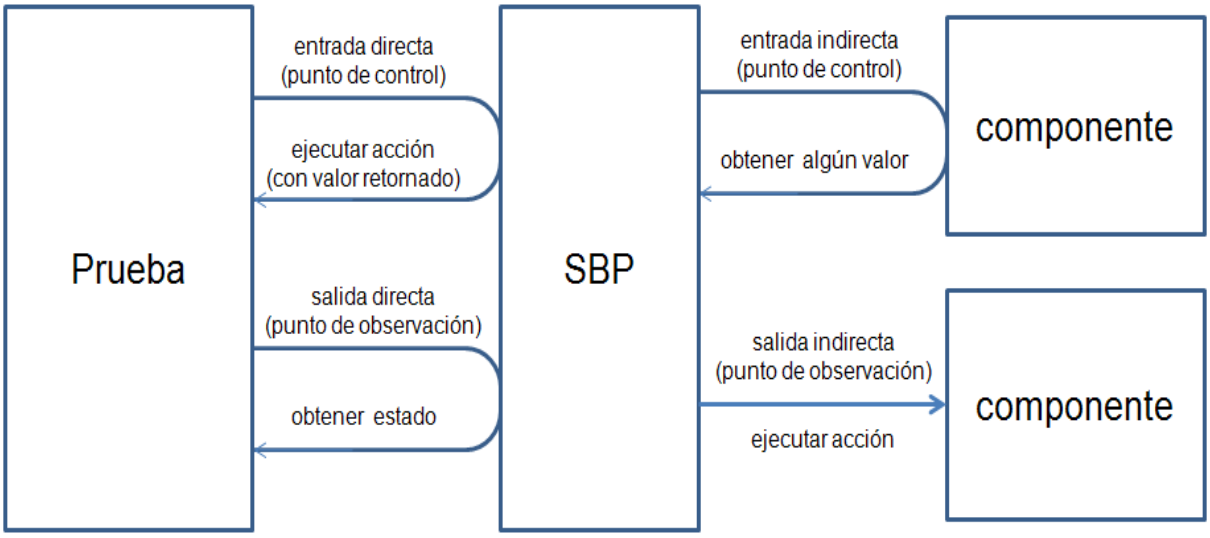

Figura 1.4 Puntos de control y puntos de observación.

¿Cómo hacer interactuar las pruebas con el SBP?

Cuando se prueba una pieza particular del software, las pruebas pueden interactuar con el SBP de diferentes maneras. Tres formas básicas pueden ser las siguientes:

- **Pruebas de ida y vuelta** – interactúan con el SBP sólo a través de sus interfaces públicas. Tanto los puntos de control como los puntos de observación son simples llamadas a métodos. La ventaja de esta manera es que no viola la encapsulación. Las pruebas solo necesitan conocer las interfaces públicas sin importar la forma en la que se construye el software.
- **Pruebas entre capas** – son la principal alternativa en la cual se ejecuta el SBP a través de las Interfaces de Aplicación del Programa (API, siglas en inglés de Application Program Interfaces) y se vigila todo por agentes dobles actuando de espías. Esta puede ser una manera muy poderosa de hacer pruebas, aunque no debería utilizarse demasiado pues los cambios en la implementación de las responsabilidades del software provocarán fallos en las pruebas.
- **Pruebas asíncronas** – las respuestas a las peticiones de las pruebas se originan asíncronamente. Estas pruebas deben incluir algún tipo de procesador interno de sincronización tal que las llamadas esperen por las respuestas. Esperar por la respuesta de los mensajes (que pueden no llegar nunca) provoca que estas pruebas tomen mucho tiempo, por lo que deben ser evitadas en pruebas unitarias y de componentes a toda costa.

1.3 Pruebas unitarias de software.

Las pruebas unitarias son un procedimiento para validar que una porción del código del sistema funciona apropiadamente de manera aislada. En caso de una programación de procedimientos se

entiende como un programa individual, una función o procedimiento, etcétera. Cuando se está programando orientado a objetos la menor de estas unidades es siempre una clase. Rob Johnson define las pruebas unitarias como el nivel más fino de granularidad en las pruebas, que verifican una simple unidad de funcionalidad y deben probar que cada método de una clase satisface su contrato documentado. (JOHNSON, ROD 2003)

Las pruebas unitarias proveen la capacidad para probar un módulo del software de forma reproducible, eficiente y automatizada. Generalmente, se organizan en suites, a manera de una colección de pruebas unitarias que tienen algo en común[22].

1.3.1 ¿Quiénes hacen las pruebas unitarias?

Para conocer quién debe responsabilizarse por crear las pruebas unitarias dentro el desarrollo del producto, tómese en cuenta los criterios siguientes:

- William E. Lewis dice: “Prueba unitaria es la más ‘micro’ escala de las pruebas; para probar funciones particulares o módulos de código. Típicamente hechas por desarrolladores y no por probadores.(…).” (LEWIS 2005).
- Rob Johnson dice:”Mientras otras formas de probar pueden ser realizadas por probadores especializados, las pruebas unitarias deben ser escritas por el desarrollador (o una pareja de estos) que escriba cada clase, junto al código de la clase en sí. Un par de programadores es particularmente más efectivo, porque dos desarrolladores pueden pensar en un rango más amplio de escenarios de pruebas que uno”.(JOHNSON, ROD and HOELLER 2004).

Si los desarrolladores escriben las pruebas en pareja no es necesario que ambos sean expertos en programar la funcionalidad que será probada. Uno de estos, incluso, puede ser un desarrollador novato que aspira a pasar al desarrollo. Una posible aplicación de esta idea, donde se pueda lograr un buen trabajo entre los dos, podría ser la siguiente:

- uno de ellos (el más experimentado en el desarrollo) sería el responsable de desarrollar el modulo y escribir las pruebas unitarias, ejecutarlas siempre que crea necesario y escribir la documentación básica;
- el otro (el menos experto) estudiaría el código y las pruebas realizadas por el anterior, realiza un análisis de las pruebas, eventualmente le dice dónde no es suficiente la cobertura, propone

[22] El agrupamiento de las pruebas en suites puede estar dado por características o propósitos comunes: ya sea que prueben el mismo módulo, que tengan configuraciones comunes o porque dependan unas de otras.

cambios en el código y las pruebas, ayuda a mantener las pruebas unitarias y las suites de pruebas, y finaliza la documentación correspondiente.

Independientemente de la forma en que sean escritas las pruebas unitarias, quien las haga debe tener las siguientes cualidades:

- Escéptico, especialmente sobre suposiciones (requiere evidencia concreta).
- Capaz de notar y seguir detalles extraños.
- Metódico y sistemático.
- Voluntad de experimentar, intentar cosas, ver lo que pasa.
- Habilidades orales y escritas que le permitan transmitir sus ideas.
- Capacidad de anticipar lo que otros no entenderán.

1.3.2 Técnicas aplicadas en las pruebas unitarias.

Anteriormente, se mencionó la manera en que las pruebas automatizadas podrían interactuar con el SBP. Según Rob Johnson (JOHNSON, ROD 2003), para el caso particular de las pruebas unitarias se pueden mencionar dos maneras principales de realizar estas pruebas al software:

- **Pruebas de caja negra**[23] - estas pruebas solo consideran las interfaces públicas de las clases bajo prueba. No se basa en el conocimiento de los detalles de implementación.
- **Pruebas de caja blanca**[24]– las pruebas son conscientes del contenido interno de las clases bajo prueba. No sólo prueba lo que es requerido hacer por la clase, sino también cómo lo hace. No es recomendable hacer pruebas de caja blanca. Estas pruebas en ocasiones son llamadas pruebas de caja de cristal.

Pruebas de Caja Negra.

Las pruebas de caja negra se centran en lo que se espera del sistema, intentan encontrar casos en que el mismo no se atiene a su especificación. Están especialmente indicadas para aquellos casos que serán interfaz con el usuario final y se apoyan en la especificación de los requisitos. Su mayor complejidad radica en los datos que se requieren para ejecutar las funcionalidades del módulo en cuestión. El conjunto de datos posibles suele ser muy amplio.

---

[23] También conocidas como pruebas de caja opaca, pruebas funcionales, pruebas de entrada/salida o pruebas inducidas por los datos.

[24] También conocidas como pruebas estructurales o pruebas de caja transparente.

Según los requisitos de un módulo, se sigue una técnica algebraica conocida como “clases de equivalencia”, donde se trata cada parámetro como un modelo algebraico donde ciertos datos son equivalentes a otros. Para ello, se parte desde un rango excesivamente amplio de posibles valores reales a un conjunto reducido de clases de equivalencia y con esto se es suficiente probar un caso de cada clase, pues los demás datos de la clase son equivalentes. Para identificar las clases de equivalencia no existe una regla universal, pero se puede tener en cuenta algunas consideraciones al respecto:

- si un parámetro de entrada debe estar comprendido en un cierto rango, aparecen 3 clases de equivalencia: por debajo, dentro y por encima del rango.
- si una entrada requiere un valor concreto, aparecen 3 clases de equivalencia: por debajo, el mismo y por encima del valor.
- si una entrada requiere un valor entre los de un conjunto, aparecen 2 clases de equivalencia: un valor contenido en el conjunto y otro fuera de él.
- si una entrada es booleana hay 2 clases: si (true) y no (false).

En la lectura de los requisitos del sistema, se pueden encontrar ciertos valores singulares que marquen diferencias de comportamiento que serán claros candidatos a marcar clases de equivalencia.

Una vez identificadas las clases de equivalencia significativas, se procede a tomar un valor de cada clase que no esté justamente en el límite (o frontera) de la clase. Además de los elegidos, es conveniente probar con los valores “frontera”. Generalmente muchos errores son detectados en torno a los puntos de cambio de las clases de equivalencia.

Lograr una buena cobertura de código con las pruebas de caja negra aseguraría que el software hace lo que se espera de él, pero no se aseguraría que haga (además) otras cosas menos aceptables.

Pruebas de Caja Blanca.

En las pruebas de caja blanca siempre se está observando el código, se conoce el contenido escrito dentro de la unidad a probar y con el ánimo de “probarlo todo” se formaliza lo que sería la “cobertura del código” a probar.

Para este tipo de pruebas existen métodos para diseñar los casos de prueba que garanticen que se ejerciten, al menos una vez, todos los caminos independientes de cada módulo, las decisiones lógicas, los ciclos en sus límites operacionales y estructuras de datos internas, y así asegurar su validez. Con estos casos de prueba se pretende demostrar que las funciones del software son operativas, aceptando correctamente los datos de entrada y produciendo los resultados esperados, donde la estructura interna se comporta de la manera requerida.

Respecto a este tipo de pruebas, Rob Johnson dice: “(…) la existencia de las pruebas promueve la refactorización, siendo posible ejecutar pruebas existentes que brinden la tranquilidad de que la refactorización no ha dañado algo. Las pruebas de caja blanca reducen el valor de este importante beneficio. Si las pruebas dependen de los detalles de implementación de una clase, refactorizar la clase tiene el potencial de romper tanto la clase como la prueba simultáneamente. Si mantener las pruebas se convierte en algo trabajoso, estas no serán mantenidas y la estrategia de probar terminará.”(JOHNSON, ROD 2003)

1.3.3 Cobertura de código en las pruebas unitarias.

La cobertura de las pruebas unitarias está dada por la medida de cuánta parte del código es probada. Significa inspeccionar si cada línea de código del SBP es ejecutada al menos una vez en alguna prueba.

Es importante conocer si las pruebas brindan buena cobertura. Es inútil -o al menos, no se podrían aprovechar al máximo todas las bondades de las pruebas- si se tiene una suite de pruebas que no prueban todo el código. Una suite de pruebas debe contar siempre con instrumentos que permitan conocer si las pruebas han olvidado algo por probar. Así, por lo menos se conocerá donde hay mayor probabilidad de que los usuarios o probadores encuentren errores.

Entre los tipos básicos de cobertura que pueden lograr las pruebas están:

- **Cobertura de segmentos** - por segmento se entiende una secuencia de sentencias sin puntos de decisión (sin estructuras condicionales), donde el ordenador está obligado a ejecutarlas secuencialmente una tras otra. En la práctica, el proceso de pruebas puede terminar antes de llegar al 100%, pues puede ser excesivamente laborioso y costoso provocar el paso por todas y cada una de las sentencias. En ocasiones puede que los programas contengan código muerto o inalcanzable, que simplemente sobre y pueda prescindirse de él.
- **Cobertura de ramas** - la cobertura de segmentos es engañosa en presencia de segmentos opcionales. Para estos casos se plantea un refinamiento de la cobertura de segmentos consistente en recorrer todas las posibles salidas de los puntos de decisión. Desde el punto de vista de la lógica del código deben verificarse casos en el que se cumpla la condición y casos en el que no se cumpla. También debe tomarse en cuenta este criterio para lenguajes que admiten excepciones.
- **Cobertura de condición/decisión** - la cobertura de ramas resulta a su vez no trivial cuando las expresiones booleanas son complejas. En estos casos se define un criterio de cobertura que fracciona las expresiones booleanas complejas en sus expresiones componentes e intenta

cubrir todos los posibles valores de cada una de ellas. No sólo basta con cubrir cada una de las condiciones componentes de una decisión, sino también todas las posibles combinaciones.

- **Cobertura de bucles** - los bucles no son más que segmentos controlados por decisiones, por lo que la cobertura de ramas cumple plenamente la esencia de la cobertura de bucles, teóricamente. En la práctica, los bucles son una fuente inagotable de errores. Un bucle se ejecuta un cierto número de veces, el cual debe ser lo más preciso posible; lo más normal es que ejecutarlo una vez de más o de menos tenga consecuencias indeseables. Al diseñar las pruebas debe procurarse que las sentencias dentro de lo bucles sean ejecutadas 0 veces, 1 vez y más de una vez.

En la práctica se suele procurar alcanzar una cobertura cercana al 100% de segmentos. Es muy recomendable (aunque cuesta más) conseguir una buena cobertura de ramas. No obstante eso, no siempre es imprescindible ir en busca de una cobertura total. En dependencia de lo crítico que sea el software debe valorarse el riesgo (costo) que implica un fallo si es descubierto durante la ejecución del sistema. Para casos menos problemáticos[25] en la consecuencia de la falla es admisible una cobertura de un 60-80%. Si la falla puede implicar pérdidas graves (por ejemplo, de dinero o vidas humanas) la cobertura se debería ser en un mayor por ciento[26].

1.3.4 ¿Cómo deben ser las pruebas unitarias en la plataforma J2EE?

Para las pruebas unitarias deben tomarse ciertos criterios en cuenta que determinen la manera en que estas serán realizadas. En el caso especifico de las pruebas unitarias en la plataforma J2EE, existen particularidades de la misma que hacen distintivas la manera de hacerlas. Algunas valoraciones descritas por Rob Johnson son las siguientes (JOHNSON, ROD 2003):

Escribir pruebas a interfaces.

Siempre que sea posible, debe escribirse las pruebas a interfaces más que a las clases. Es una buena práctica de diseño orientado a objeto. Así diferentes suites de pruebas se podrán crear fácilmente para ser ejecutadas contra las implementaciones de una interface.

---

[25] Ejemplo de esto puede ser en video juego, donde la falla sólo provoca que deba empezarse nuevamente a jugar; no así en una aplicación bancaria donde podría provocar pérdidas millonarias de dinero.

[26] En aplicaciones militares, por ejemplo, se perseguiría más de un 90% de cobertura.

No probar propiedades de JavaBeans[27].

Usualmente es innecesario probar los métodos de tipo **`getXXX()`** y **`setXXX()`** que obtienen y modifican los valores de los atributos en una clase, respectivamente. Se malgasta tiempo de desarrollo en tales pruebas. Además, aumentar las pruebas con códigos que no son útiles las hace más difícil de leer y mantener.

Maximizar la cobertura de las pruebas.

Escribir las pruebas antes que el código de producción es la mejor estrategia para maximizar la cobertura de las pruebas. Algunas herramientas pueden ayudar a verificar que se alcanza este propósito.

No confiar en el orden de las pruebas.

Cuando se identifican los métodos a ejecutar, no se garantiza el orden en el cual estos serán ejecutados. Por lo que las pruebas no deben tener en cuenta algún orden en el que pudieran ejecutarse previamente. En caso de que sea vital tener alguna prioridad entre ellos, esto sería posible si se adicionan a una suite de pruebas de manera explícita y estos serán ejecutados en el orden en que fueron adicionados.

Evitar efectos colaterales.

Por las mismas razones, es importante evitar efectos colaterales cuando se prueba. Esto ocurre cuando una prueba cambia el estado del sistema cuando es probado de modo que puede afectar subsiguientes pruebas. Cambios a los datos en una base de datos son un ejemplo potencial de efectos colaterales.

Leer datos de prueba localizados en el Class-Path[28], no desde el File System[29].

Muchas pruebas necesitan ser configuradas para su ejecución. En ocasiones leen su configuración desde el sistema de archivos. Usar caminos absolutos de ficheros puede provocar problemas cuando el código es probado en otra localización. Esto puede hacer –entre otras cosas- que las pruebas

---

[27] Los JavaBeans son clases cuyo único contenido son sus atributos y métodos del tipo getXXX() y setXXX(), que no implementan ninguna funcionalidad con comportamiento complejo.

[28] El Class-Path es una variable que indica la estructura de recursos residentes en el directorio de compilación y ejecución del sistema.

[29] El File System indica el directorio de archivos del sistema operativo y de la estación de trabajo.

dependan de un sistema operativo o la estructura de archivos de alguna estación de trabajo. Este problema puede ser evitado obtenido los datos de configuración relativamente al código. Los recursos necesarios son usualmente colocados en el mismo directorio que las clases que los usan.

Evitar duplicación en las pruebas.

Al igual que el código de la aplicación, mientras más código duplicado exista en las pruebas más posibilidades de contener errores tendrán. En muchas pruebas la mayoría de su código puede ser compartido por otras. El uso de métodos útiles que puedan ser reutilizados simplifica el código de los métodos de prueba.

¿Cuándo se deben crear agentes dobles?

En ocasiones clases a ser probadas dependen de otras clases que no están disponibles o no han sido implementadas aún en el momento requerido[30]. En las aplicaciones J2EE, es común que tales dependencias sean a clases provistas por el servidor de aplicaciones[31] y que se desee probar la clase aislada del servidor J2EE. En tales casos, se pueden escribir simples implementaciones de las interfaces requeridas por las clases bajo prueba, lo cual tiene real valor cuando no implican mucho esfuerzo; es mejor evitar escribirlas cuando sean complejas.

Herencia en las pruebas.

Se deben considerar las implicaciones de la herencia de clases que son probadas. Estas clases deben pasar todas las pruebas asociadas con sus superclases y las interfaces que implementen. Por otro lado, cuando una clase de prueba herede directamente de otra clase de prueba, todos los métodos de prueba de la superclase serán ejecutados. Cuando se escriba una prueba para interfaces o clases abstractas, la clase de pruebas puede ser abstracta y hacer que las pruebas a las subclases (o a las implementaciones concretas) sean probadas independientemente en clases de pruebas heredadas.

¿Dónde deben estar colocadas las clases de prueba?

Las clases de prueba deben estar separadas del código a ser probadas. Se evita, entre otras cosas, generar los Javadoc de las pruebas y sería más fácil tenerlas fuera de la estructura de las clases de la aplicación. Una buena práctica sería usar paquetes paralelos a la estructura de las clases a ser

---

[30] Si se siguen las buenas prácticas anteriores, las dependencias serán a interfaces, y no se dispondría de implementaciones a ellas.

[31] Por ejemplo, una clase que requiere un objeto javax.sql.DataSource que proveen de conexiones a un servidor de bases de datos, el cual puede no estar disponible en el momento de la prueba.

probadas en una carpeta de código distinta. Esto permite acceder a los atributos y métodos protegidos de las clases y hace más fácil encontrar las clases de prueba de cualquier clase del sistema.

1.4 Conclusiones.

Los desarrolladores deben trazarse propósitos en las pruebas que puedan alcanzar durante el desarrollo del producto. Y aunque no siempre es posible que todos adopten una misma filosofía de trabajo, debe llegarse a un entendimiento de por qué se hacen las cosas de manera diferente y procurarse que se compartan los mismos propósitos, tales como la alta calidad en el software, el cumplir un fin específico, terminar en tiempo, estar por debajo del presupuesto, entre otros. Justamente estos criterios son los que crean las decisiones de cómo alcanzar los objetivos.

De manera general, las pruebas quedaran por sí mismas documentadas, ayudaran a reducir riesgos y mejorarán la calidad del producto. Deben ser fáciles de escribir y mantener, expresivas, completamente automatizadas, repetibles, robustas y que no provoquen daños al sistema. Se deben escribir tantas pruebas cómo sean posibles y necesarias, a partir de los criterios expuestos.

Estas son un proceso dentro del desarrollo del software con la intención de descubrir errores, sin embargo, no se debe estar totalmente confiado, porque no se puede asegurar totalmente la ausencia de los mismos.

CAPÍTULO 2: HERRAMIENTAS Y TÉCNICAS DE PRUEBAS UNITARIAS EN LA PLATAFORMA J2EE.

2.1 Introducción.

Además de todas las características que deben cumplir las pruebas unitarias[32], es importante conocer sus especificidades dado un lenguaje particular. Las ventajas o desventajas de la plataforma en la que se construya el software, influye considerablemente en las estrategias que se tracen los desarrolladores para realizarlas. Los tipos de componentes que se crean también pueden estar vinculados directamente a la forma en que estos puedan ser probados.

Debido a la inherente complejidad de los softwares desarrollados en la plataforma J2EE, las pruebas pueden ser más difíciles de lo que suelen parecer. Con J2EE se aplica un gran número de tecnologías tales como Java Server Pages (JSP), Servlets, Enterprise JavaBeans (EJBs), acceso a bases de datos (JDBC), entre otras, por lo que se suelen requerir varias técnicas y herramientas de prueba. En el caso de las pruebas unitarias en J2EE, existe diversidad de herramientas que implementan técnicas y aplican estrategias generales. Otras ofrecen soluciones para componentes específicos que en ciertos momentos puede ser ideal su selección.

En este capítulo se instruye en el trabajo con herramientas utilizadas en la realización de pruebas unitarias para aplicaciones en J2EE apoyado en ejemplos que vinculan buenas prácticas a diferentes métodos de trabajo. Se proponen además herramientas que pueden servir de apoyo en el proceso de automatización y obtención de resultados de las pruebas unitarias.

2.2 Frameworks de pruebas unitarias en J2EE.

2.2.1 Patrón xUnit.

Las pruebas escritas a mano generalmente son automatizadas usando el mismo lenguaje que se utiliza en construir el SBP puesto que es más viable el acceso a las mismas API que el sistema. Esto no es obligatorio Usar un lenguaje familiar para los desarrolladores disminuye el esfuerzo requerido para aprender cómo automatizar completamente las pruebas. Para ello, muchos frameworks son implementados basados en el patrón xUnit usando lenguajes de programación orientados a objetos. El término xUnit se refiere a cualquier miembro de la familia de frameworks de pruebas automatizadas. La

[32] A partir de aquí, siempre que se mencionen “las pruebas” se estarán indicando las pruebas unitarias, a no ser que se especifique lo contrario en el contexto en que se enmarque.

mayoría de los lenguajes de programación, hoy cuentan con al menos una implementación de xUnit, donde se comparten características básicas que permiten lograr las siguientes funcionalidades:

- Especificar las pruebas como “métodos de prueba”.
- Especificar los resultados esperados dentro del método de prueba en forma de llamadas a “métodos de aserción”.
- Agregar las pruebas en “suites de pruebas” que puedan ser ejecutadas como una sola operación.
- Ejecutar una o más pruebas para obtener un reporte de los resultados de la ejecución de las pruebas.

Algunos de los frameworks xUnit tienen soporte para descubrir los métodos de prueba o alguna forma de selección para ejecutar subconjuntos de métodos de prueba basados en algún criterio.

2.2.2 Framework JUnit.

En el caso de aplicaciones basadas en J2EE, la elección estándar es JUnit, framework de código abierto para pruebas en Java. Originalmente escrito por Kent Beck[33] y Erich Gamma[34] .Se integra con muchos Entornos de Desarrollo Integrado (IDE, siglas en inglés de Integrated Development Environment) como NetBeans, BlueJ, IntelliJ, JBuilder y Eclipse. Ha sido modelo inspirador de muchos frameworks xUnit como NUnit para .NET, PyUnit para Python, CppUnit para C++, DUnit para Delphi, entre otros. De hecho, la idea de los patrones xUnit en sí fue tomada del framework JUnit.

Con JUnit se logra ejecutar pruebas automatizadas fácilmente. Contiene mecanismos para recolectar resultados de manera estructurada, produce varios tipos de reportes a partir de los resultados obtenidos en la ejecución de las pruebas, permite que existan relaciones entre las pruebas y que unas reutilicen código de otras. Además, soporta la jerarquía entre las pruebas pudiendo establecerse la prioridad y el orden de unas con otras.

Entre las características principales de JUnit están (Figura 2.1):

- API para crear clases y métodos de pruebas (**`TestCase`**, **`testXXX`**).
- Configuraciones en las pruebas para compartir datos de prueba comunes (**`setUp`**, **`tearDown`**).

---

[33] Kent Beck, creador y pionero de la metodología XP.

[34] Erich Gamma, prestigioso en la literatura de Patrones de Diseño, es uno de los “Gang of Four”.

- Fácil comprensión de las aserciones para los resultados esperados de las pruebas (**`assertXXX`**).
- Ejecutores de pruebas de manera gráfica y textual (**`TestRunner`**).
- Suites de pruebas para organizar y ejecutar las pruebas. (**`TestSuite`**)
- Reportes de los resultados (**`TestResult`**).

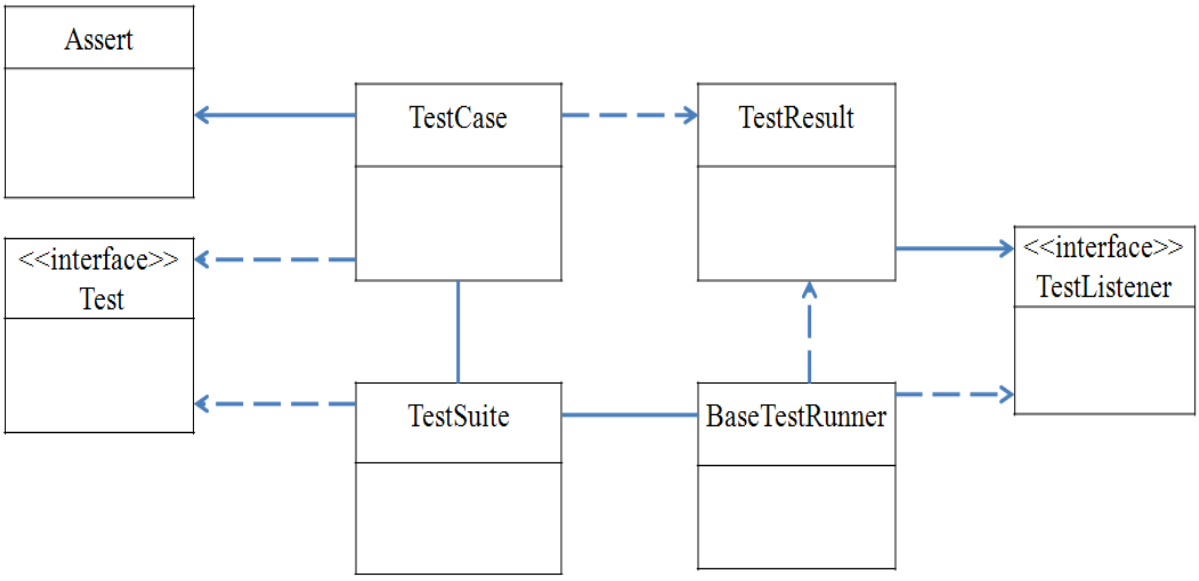

Figura 2.1 Clases del núcleo de JUnit utilizadas en la ejecución de las pruebas.

JUnit alienta a los desarrolladores a escribir pruebas unitarias gracias a la variedad de patrones en los que se abstrae para facilitar su comprensión y, con ello, su uso. Martin Fowler dijo refiriéndose a JUnit en el prólogo del libro xUnit Test Patterns: ”(…) nunca en el campo del desarrollo de software hubo algo así a lo que deberle tanto por tan pocas líneas de código (…).”(MESZAROS 2007).

Los pasos típicos para escribir código de pruebas basados en JUnit son:

- Crear una clase de prueba que sea subclase de **`TestCase`**.
- Crear un método nombrado de la forma **`testXXX()`** para cada prueba individual a ser ejecutada.
- Crear, si es necesario, un método nombrado **`setUp()`** para la configuración de los recursos de apoyo a la ejecución de la prueba.
- Escribir la prueba y comprobar el comportamiento que interese con los métodos nombrados de la forma **`assertXXX()`**.
- Crear, de ser necesario, un método nombrado **`tearDown()`** para restablecer los cambios ocasionados por la configuración anterior.

- Opcionalmente, crear una suite para organizar las pruebas con instancias de la clase **`TestSuite`**.
- Ejecutar las pruebas con interfaces de texto en consola o interfaces visuales en algún IDE.

Clase TestCase y métodos testXXX().

La clase **`TestCase`**, es usada para crear las clases de pruebas. Define el armazón del algoritmo básico de las pruebas, delegando en las subclases correspondientes la posibilidad de implementar la manera en que sucederán esos pasos.

Para escribir el código de prueba, primeramente se debe definir una subclase de **`TestCase`**, que suele nombrársele igual a la clase o funcionalidad a probar, agregándole la terminación **`Test`**. Se recomienda colocarla dentro del mismo paquete de clases, pero en otra jerarquía de carpeta de código, lo que hará más fácil el determinar cuál es la clase que implementa la prueba de alguna clase del código de producción y viceversa. De esta manera se podrán acceder a los métodos y atributos protegidos de la clase y a la vez se separa el código de prueba del código de producción. Por cada prueba que se quiera realizar se implementa un método público que no devuelva resultado (**`void`**) ni reciba parámetros. Su nombre debe comenzar con **`test`** y el resto del mismo podría describir la funcionalidad que será probada. En la Figura 2.2 se muestra un ejemplo de ello.

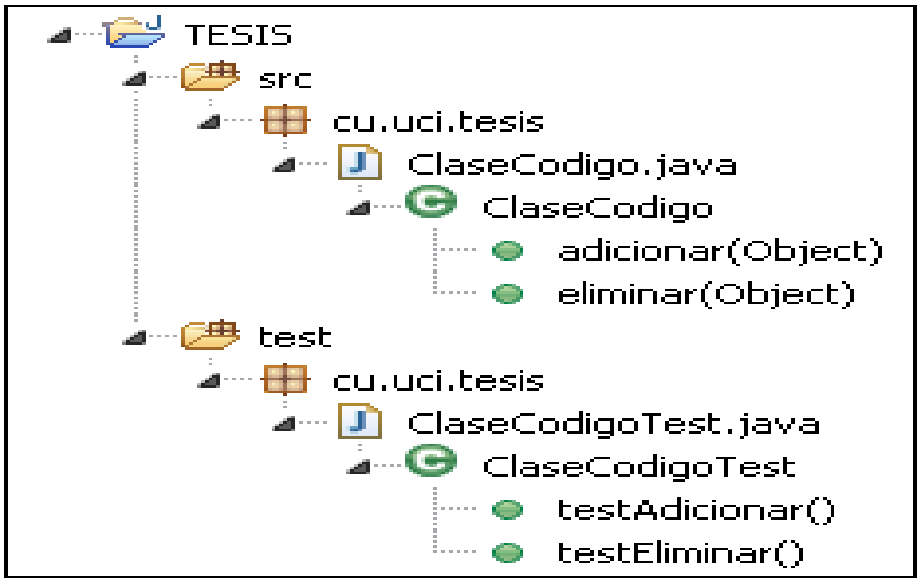

Figura 2.2 Jerarquía recomendada para la creación y localización de las clases de prueba.

En cada método **`testXXX()`** de la clase de prueba, se escribirán sentencias de las operaciones del SBP que se deberán ejecutar y las aserciones a través de llamadas a métodos **`assertXXX()`** correspondientes a las verificaciones que se deseen hacer.

*** *Ejemplo***: Supóngase tener una clase **`RangoNumerico`** que tiene atributos **`minimo`** y **`maximo`**, indicando los valores extremos en el rango de números enteros. En tal clase se desea implementar un método nombrado **`sumarTodosEnIntervalo()`** que devuelve la suma de los números enteros comprendidos en el rango numérico. En el código 2.1 se muestra la implementación de una clase de prueba para verificar el método descrito en la clase dada.

Código 2.1 – Ejemplo de una clase de prueba basada en JUnit.

```
package cu.uci.tesis.capitulo2.testcase;
import junit.framework.TestCase;
public class RangoNumericoTest extends TestCase{
      public void testSumarTodosEnIntervalo( ){
            RangoNumerico rango=new RangoNumerico (4, 10);
            int actual=rango.sumarTodosEnIntervalo();
            int esperado=4+5+6+7+8+9+10;
            assertEquals( esperado, actual);
      }
}
```

En el ejemplo anterior se puede ver cómo las pruebas pueden servir de especificación y documentación del sistema. Cuando el programador que debe implementar el método **`sumarTodosEnIntervalo()`** no esté seguro si debe incluir o no los extremos del intervalo a la hora de efectuar la suma, con un simple vistazo al código de la prueba estará completamente fuera de dudas.

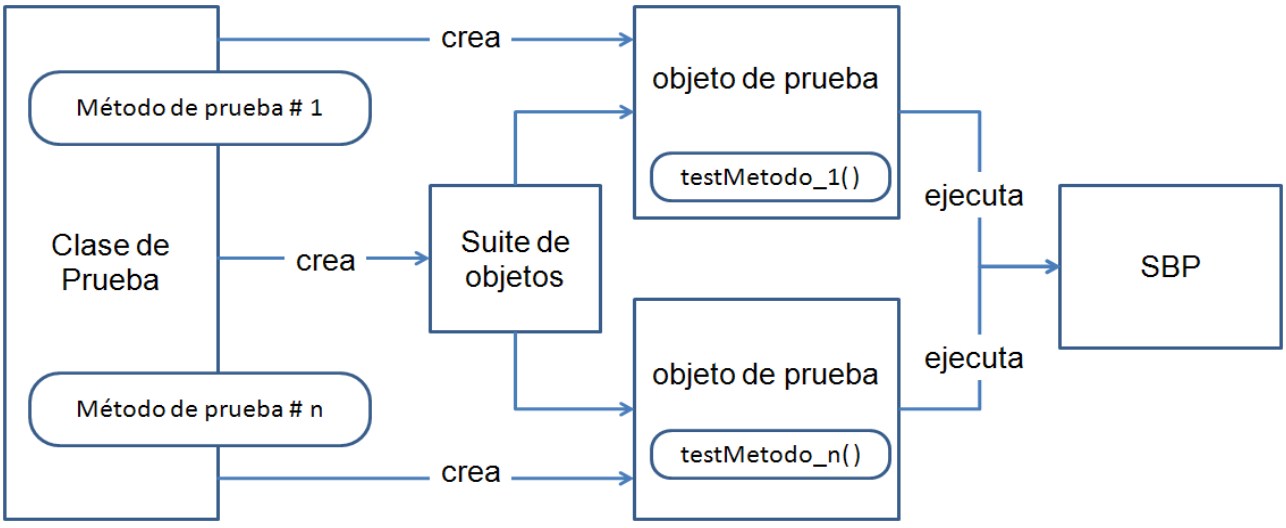

Figura 2.3 Suite de objetos de prueba. Cada objeto ejecutará uno solo de los métodos de prueba.

Cuando se ejecuta una clase de prueba que contiene varios métodos de prueba, se crea una suite de objetos de prueba, uno por cada método de prueba. En la Figura 2.3 se muestra cómo cada objeto representante de la clase será ejecutor de un único método de prueba, de manera aislada a los demás.

Clase TestSuite y método suite().

En ocasiones se tiene una clase de prueba con más de un método de prueba, los cuales serán ejecutados en algún orden. Este orden no es siempre el mismo o no puede asegurarse a priori cuál será. Aunque se recomienda que los métodos de pruebas no sean orden-dependientes, a veces puede que sea necesario que la ejecución de uno espere por la finalización de otro. Por otro lado, puede que no se desee que todos los métodos de una prueba se ejecuten en la misma suite de objetos de prueba. El caso puede darse cuando se quiere seleccionar un conjunto de métodos de pruebas que chequeen un componente determinado y no se quiera emplear tiempo extra en los métodos que chequearán otras funcionalidades y que inevitablemente se encuentran en la misma clase de prueba.

A través de la clase **`TestSuite`** es posible lograr estos propósitos. Creando una suite de pruebas se puede determinar cuáles métodos de pruebas se desean ejecutar y en qué orden, según se agregan explícitamente a la suite. Según se muestra en la Figura 2.4, esto permite crear una jerarquía de pruebas donde se deciden qué clases y métodos de pruebas serán ejecutados y cuál será el orden que tendrán durante su ejecución.

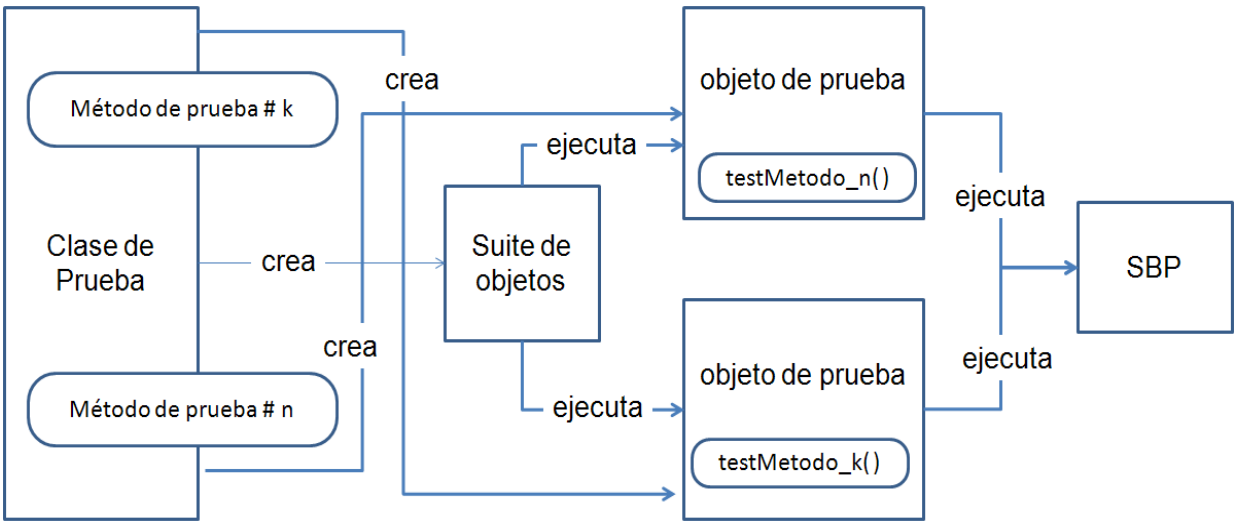

Figura 2.4 En una suite de pruebas se puede indicar los métodos de pruebas a ejecutar y el orden.

*** *Ejemplo***: Teniendo en cuenta el ejemplo anterior, a la clase **`RangoNumerico`** se le agregan los métodos **`hallarCantidadDivisiblesPor(int)`** para hallar la cantidad de números divisibles por un valor dado y **`estaContenido(int)`** para determinar si un valor dado se encuentra en un intervalo numérico. En la clase de prueba correspondiente se desea establecer un orden específico

para la ejecución de cada uno de los métodos de prueba. En el código 2.2 se muestra la primera de las variantes posibles, creando un método **`suite()`** dentro de la misma clase de prueba.

Código 2.2 – Suite de pruebas indicando el orden a través del método suite( )

```
package cu.uci.tesis.capitulo2.suite;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class RangoNumericoTest extends TestCase {
      public RangoNumericoTest(String name) {
            super(name);
      }
      public void testSumarTodosEnIntervalo() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            int actual = rango.sumarTodosEnIntervalo();
            int esperado = 4 + 5 + 6 + 7 + 8 + 9 + 10;
            assertEquals(esperado, actual);
      }
      public void testHallarCantidadDivisibles() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            int actual = rango.hallarCantidadDivisiblesPor(3);
            int esperado = 2;
            assertEquals(esperado, actual);
      }
      public void testEstaContenido() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            boolean actual = rango.estaContenido(6);
            boolean esperado = true;
            assertEquals(esperado, actual);
      }
      public static TestSuite suite() {
            TestSuite suite = new TestSuite();
            suite.addTest(new
                        RangoNumericoTest("testHallarCantidadDivisibles"));
            suite.addTest(new
                           RangoNumericoTest("testSumarTodosEnIntervalo"));
```

```
            suite.addTest(new RangoNumericoTest("testEstaContenido"));
            return suite;
      }
}
```

Para adicionar un método específico de una clase de prueba a una suite de objetos de prueba, dicha clase debe tener un constructor que reciba como parámetro el nombre del método en cuestión. De esta manera se le indica a la suite de objetos de prueba, cuál sería el método a ser ejecutado. A una suite se le pueden agregar también suites completas de una clase de prueba, indicando solo cuál es esta clase. En el código 2.3 se muestra un ejemplo de ello donde el resultado sería similar si no se hubiese implementado el método **`suite()`**.

Código 2.3 – Suite de pruebas indicando la suite completa de una clase

```
package cu.uci.tesis.capitulo2.suite;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class RangoNumericoTest extends TestCase {
      public RangoNumericoTest(String name) {
            super(name);
      }
      public void testSumarTodosEnIntervalo() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            int actual = rango.sumarTodosEnIntervalo();
            int esperado = 4 + 5 + 6 + 7 + 8 + 9 + 10;
            assertEquals(esperado, actual);
      }
      public void testHallarCantidadDivisibles() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            int actual = rango.hallarCantidadDivisiblesPor(3);
            int esperado = 2;
            assertEquals(esperado, actual);
      }
      public void testEstaContenido() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            boolean actual = rango.estaContenido(6);
```

```
            boolean esperado = true;
            assertEquals(esperado, actual);
      }
      public static TestSuite suite() {
            TestSuite suite = new TestSuite();
            suite.addTestSuite(RangoNumericoTest.class);
            return suite;
      }
}
```

La otra de las variantes, es crear otra clase de prueba, ya sea subclase de **`TestCase`** o de **`TestSuite`**, donde se implementaría el método **`suite()`** agregándosele las suites de otras clases de prueba. En el ejemplo mostrado en el código 2.4 se puede ver, la forma más general de crear suites de pruebas. Ya sea agregando métodos, clases u otras suites de prueba.

Código 2.4 – Suite de pruebas agregando suites de otras clases de prueba.

```
package cu.uci.tesis.capitulo2.suite;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class RangoNumericoTestSuite extends TestCase {
      public TestSuite suite( ) {
            TestSuite suite = new TestSuite( );
            suite.addTest(new
                        RangoNumericoTest("testHallarCantidadDivisibles"));
            suite.addTestSuite(RangoNumericoTestSuite.class);
            suite.addTest(RangoNumericoTest.suite( ));
            return suite;
      }
}
```

Con las suites de pruebas se logran ejecutar todas las pruebas deseadas en un momento determinado, estableciendo una jerarquía para las mismas. De esta forma, seleccionar las pruebas a ejecutar no sería un proceso manual sino un aspecto declarado explícitamente en el código de la prueba.

Métodos setUp() y tearDown().

En el capítulo anterior se mencionaron las cuatro etapas (o fases) fundamentales con las que contaban las pruebas: establecer configuración, ejecución del SBP, verificación de resultados y desmontar configuración, según se muestra en la Figura 2.5.

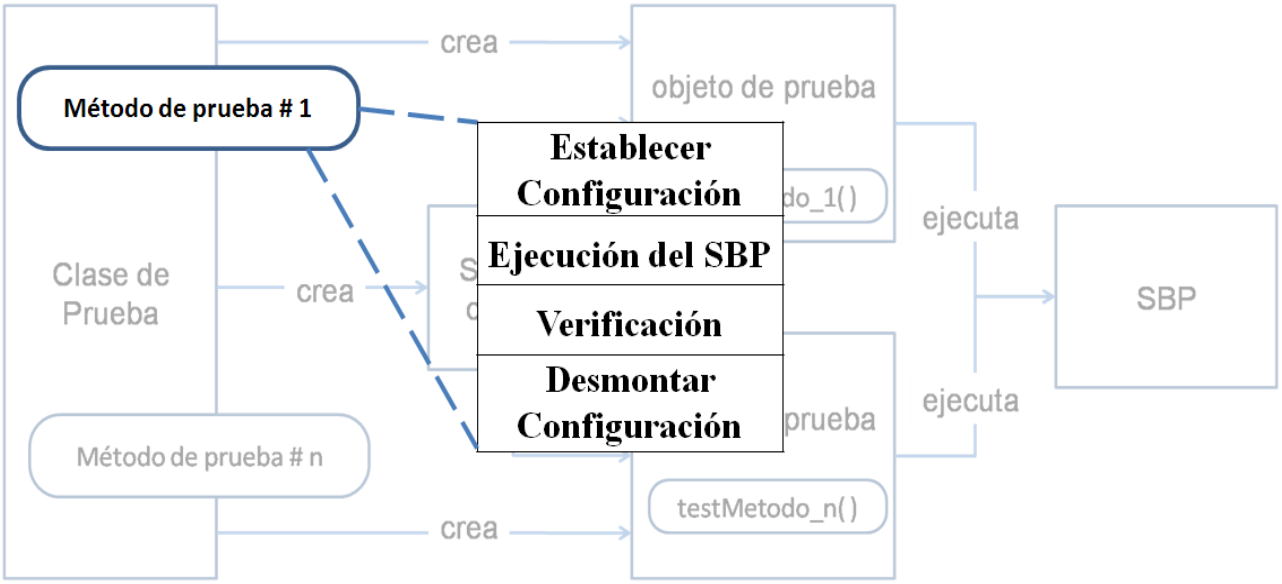

Figura 2.5 Fases en el proceso de las pruebas automatizadas

En la primera de estas fases es donde el programador debe colocar el SBP en el estado necesario para ejecutar las pruebas. La última de estas fases le permite restablecer el estado del SBP para que otras pruebas puedan funcionar correctamente, sin necesidad de conocerse entre sí. Los métodos **`setUp()`** y **`tearDown()`** son usados para inicializar y liberar la carga de los datos que son compartidos en común por estas pruebas.

El método **`setUp()`** se ejecuta antes de la invocación de cada método de prueba y el método **`tearDown()`** después de que es finalizado. Cada método de prueba implica una instancia de la clase de prueba en la suite de prueba, por lo que en cada uno de estos objetos se invocará los métodos **`setUp()`**, el correspondiente **`testXXX()`** y el **`tearDown()`**, en ese orden.

*** *Ejemplo***: Supóngase tener una clase **`Diccionario`** que tiene como atributo una lista de todas las palabras que deban ser registradas en la aplicación. Este atributo es estático para que sea único en todo el sistema. En la misma existen dos métodos: uno que obtiene una lista con las palabras ordenadas y otro que obtiene una lista con todas las palabras que contienen cierta letra. Para probar estos dos métodos que comparten una fuente de datos en común debe procurarse que la ejecución de uno no implique conocer el estado del sistema luego de la ejecución del otro. Asimismo, debe asegurarse que cada prueba sea responsable de poner el sistema en el estado inicial en el que se

encontraba antes de su ejecución. En el código 2.5 se muestra un ejemplo de cómo lograr esto en una clase de prueba.

Código 2.5 – Clase DiccionarioTest usando métodos setUp() y tearDown().

```
package cu.uci.tesis.capitulo2.setupteardown;
import java.util.List;
import junit.framework.TestCase;

public class DiccionarioTest extends TestCase {
      private Diccionario diccionario;
      public void setUp() {
            diccionario = new Diccionario();
            diccionario.adicionarPalabra("setup");
            diccionario.adicionarPalabra("teardown");
      }
      public void tearDown() {
            diccionario.eliminarTodas();
      }
      public void testOrdenarAlfabeticamente() {
            diccionario.adicionarPalabra("junit");
            List<String> ordenadas = diccionario.ordenarPalabras();
            assertEquals("junit", ordenadas.get(0));
            assertEquals("setup", ordenadas.get(1));
            assertEquals("teardown", ordenadas.get(2));
      }
      public void testPalabrasQueContienenU() {
            diccionario.adicionarPalabra("pruebas");
            List<String> contenedoras = diccionario.
                                                palabrasQueContienen('u');
            assertEquals(2, contenedoras.size( ));
            assertEquals("setup", contenedoras.get(0));
            assertEquals("pruebas", contenedoras.get(1));
      }
}
```

La implementación de los métodos **`setUp()`** y **`tearDown()`** no sólo le quitan la responsabilidad a los métodos de prueba de establecer un estado inicial en el SBP sino que también los abstraen de la

responsabilidad de retornarlo a su estado anterior, así las pruebas se enfocan solo en lo que deben probar y son más legibles y robustas. De no haberlos implementado, los métodos de prueba deberían encargarse de la configuración del sistema tanto antes como después de ejecutarse.

Métodos assertXXX().

Existen dos conceptos relacionados con la etapa de verificación que en algunos contextos pueden tener significados similares y puedan confundirse para el caso de las pruebas, donde no sucede así. Es importante ser consciente de la diferencia que hay entre estos:

- **Aserción** – es una verificación donde se produce una falla anticipada y que es emitida con total intención de dar a conocer el motivo por el cual se ha efectuado.
- **Error** – es una falla no anticipada e inesperada. Son emitidas en tiempo de ejecución por un mal procesamiento de alguna estructura o por una condición no deseada en el contexto de la ejecución, entre otras causas.

Las aserciones en JUnit son llamadas a métodos nombrados de la forma **`assertXXX()`**, dentro de los métodos de prueba para establecer las verificaciones necesarias durante la ejecución de las pruebas, según se muestra en la Figura 2.6. Estos determinan si una prueba es exitosa o no, encapsulando el mecanismo que causa la falla de las pruebas. De manera general los **`assertXXX()`** efectúan una comparación entre un valor actual y un valor esperado.

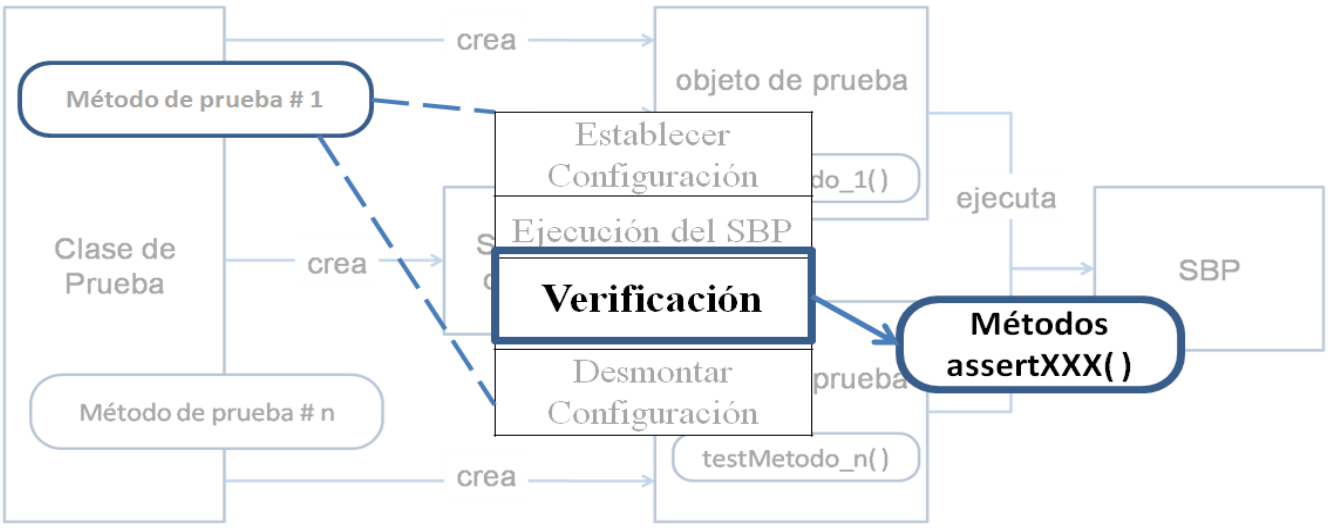

Figura 2.6 Las aserciones son llamadas a métodos **`assertXXX()`** en la etapa de verificación.

Entre los principales métodos de aserción con que cuenta la clase **`TestCase`** están:

- **`assertTrue()`** – verifican que una condición (u objeto) tiene valor **`true`**.
- **`assertFalse()`** - verifican que una condición (u objeto) tiene valor **`false`**.

- **`assertEquals()`** – verifican que dos objetos sean iguales.
- **`assertNotNull()`** – verifican que un objeto no sea **`null`**.
- **`assertNull()`** – verifican que un objeto sea **`null`**.

*** *Ejemplo***: Supóngase tener la clase **`RangoNumerico`**, con tres funcionalidades a probar: la suma de todos los elementos del intervalo, la cantidad de números divisibles por un número dado y verificar si un número está contenido en el intervalo. El código 2.6 muestra una clase de prueba donde se usan diferentes tipos de métodos **`assertXXX()`**.

Código 2.6 – Clase RangoNumericoTest usando distintos métodos assertXXX( )

```
package cu.uci.tesis.capitulo2.assertxxx;
import junit.framework.TestCase;

public class RangoNumericoTest extends TestCase {
      public void testSumarTodosEnIntervalo() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            int actual = rango.sumarTodosEnIntervalo();
            int esperado = 4 + 5 + 6 + 7 + 8 + 9 + 10;
            assertEquals(esperado, actual);
      }
      public void testHallarCantidadDivisibles() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            int actual = rango.hallarCantidadDivisiblesPor(3);
            int esperado = 2;
            assertEquals(esperado, actual);
      }
      public void testEstaContenido() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            boolean actual = rango.estaContenido(6);
            assertTrue(actual);
      }
}
```

Cuando ocurre la falla para alguna de estas verificaciones, se colecciona la descripción de la misma y es mostrada de la siguiente forma:

`junit.frameowrk.AssertionFailedError: expected: <49> but was: <35>`

Esta descripción no es totalmente clara del motivo real por el cual se produce la falla. Para estos casos la clase **`TestCase`** cuenta con métodos **`assertXXX()`** que reciben otro parámetro que indica el mensaje que se mostrará cuando la falla tenga lugar. En el código 2.7 se muestra cómo buenos mensajes de errores hacen posible encontrar la causa concreta de la falla de una prueba y permite entender el porqué de la misma.

Código 2.7 – Clase RangoNumericoTest usando distintos métodos assertXXX( )

```
package cu.uci.tesis.capitulo2.assertxxx;
import junit.framework.TestCase;

public class RangoNumericoTest extends TestCase {
      public void testSumarTodosEnIntervalo( ) {
            RangoNumerico rango = new RangoNumerico(4, 10);
            int actual = rango.sumarTodosEnIntervalo( );
            int esperado = 4 + 5 + 6 + 7 + 8 + 9 + 10;
            assertEquals("La suma 4 debe ser 49", esperado, actual);
      }
      public void testHallarCantidadDivisibles( ) {
            RangoNumerico rango = new RangoNumerico(4, 10);
            int actual = rango.hallarCantidadDivisiblesPor(3);
            int esperado = 2;
            assertEquals("El total debe ser 2", esperado, actual);
      }
      public void testEstaContenido() {
            RangoNumerico rango = new RangoNumerico(4, 10);
            boolean  = rango.estaContenido(6);
            assertTrue("El 6 se encuentra entre 4 y 10", );
      }
}
```

Cuando en el código de un método de prueba existan sentencias con llamadas a métodos que puedan lanzar excepciones no deseadas es necesario hacerlas dentro de un bloque **`try{…}catch(…){…}`** y tratarlas con llamadas a métodos **`fail()`** pasándole como parámetro el mensaje del error de la excepción.

*** *Ejemplo***: Supóngase tener una clase **`Calculadora`** donde se debe implementar un método que calcule el valor inverso de un número dado. Este método debe lanzar una excepción cuando dicho valor sea cero. La clase **`CaculadoraTest`** en el código 2.8 muestra dos casos distintos de tratar con métodos que, en algún momento, pueden lanzar excepciones.

Código 2.8 – Clase CalculadoraTest con situaciones distintas donde usar el método fail( )

```
package cu.uci.tesis.capitulo2.fail;
import junit.framework.TestCase;

public class CalculadoraTest extends TestCase {
      public void testCalcularInversoOk() {
            try {
                  float actual = Calculadora.calcularInverso(2);
                  float esperado = 0.5f;
                  assertEquals("El inverso es 0.5", esperado, actual);
            } catch (Exception error) {
                  fail(error.getMessage( ));
            }
      }
      public void testCalcularInversoException() {
            try {
                  Calculadora.calcularInverso(0);
                  fail("Se debió lanzar una excepción");
            } catch (Exception error) {
                  assertNotNull("Debía lanzarse excepción", error);
            }
      }
}
```

En el primero de los métodos de prueba, **`testCalcularInversoOK()`**, no se espera que se lance una excepción: de suceder sería un error “inesperado”, se capturaría y se le aplica el método **`fail()`**. En el segundo caso, del método de prueba **`testCalcularInversoException()`**,sí se espera que se lance una excepción: de no suceder, entonces este sería el error “inesperado” por tanto se obliga a fallar al método de prueba.

Clase TestRunner.

Durante la ejecución de las pruebas, alguien se debe encargar de llevar todo el proceso a cabo y recolectar los resultados brindados por las pruebas. En la Figura 2.7 se muestra el funcionamiento de la clase **`TestRunner`**, la cual es la encargada de hacer esta tarea.

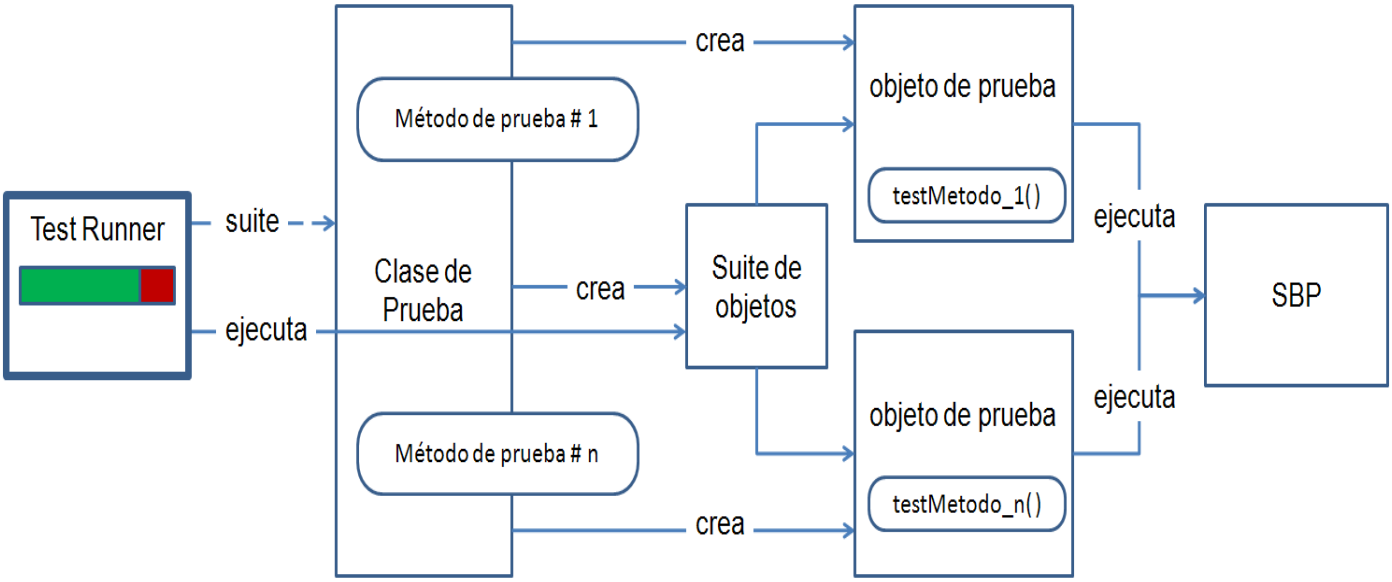

Figura 2.7 El TestRunner ejecuta la suite de objetos de prueba y dirige todo el proceso de ejecución.

Las clases **`junit.awtui.TestRunner`** y **`junit.swingui.TestRunner`** brindan interfaces gráficas para ejecutar las pruebas, basadas en **`AWT`** y **`Swing`** respectivamente. En el caso de la clase **`junit.textui.TestRunner`** cuenta con una herramienta basada en la línea de comandos. En el código 2.9 se muestra cómo ejecutar una suite de pruebas y que el resultado sea visualizado en alguna de estas interfaces, donde basta crear un método **`main()`** y ejecutar el método **`run()`** de cualquiera de las clases **`TestRunner`**, según la interfaz visual o de texto con la cual desee interactuar. La figura 2.8 muestra cómo se verían cada una de estas interfaces al ser ejecutadas las clases **`TestRunner`** correspondientes.

Código 2.9 – Clase TestRunnerAllTest que ejecuta una suite de prueba.

```
package cu.uci.tesis.capitulo2.testrunner;
import junit.swingui.TestRunner;
import cu.uci.tesis.capitulo2.fail.CalculadoraTest;

public class TestRunnerAllTest {
     public static void main(String[] args) {
```

```
            TestRunner.run(CalculadoraTest.class);
        }
}
```

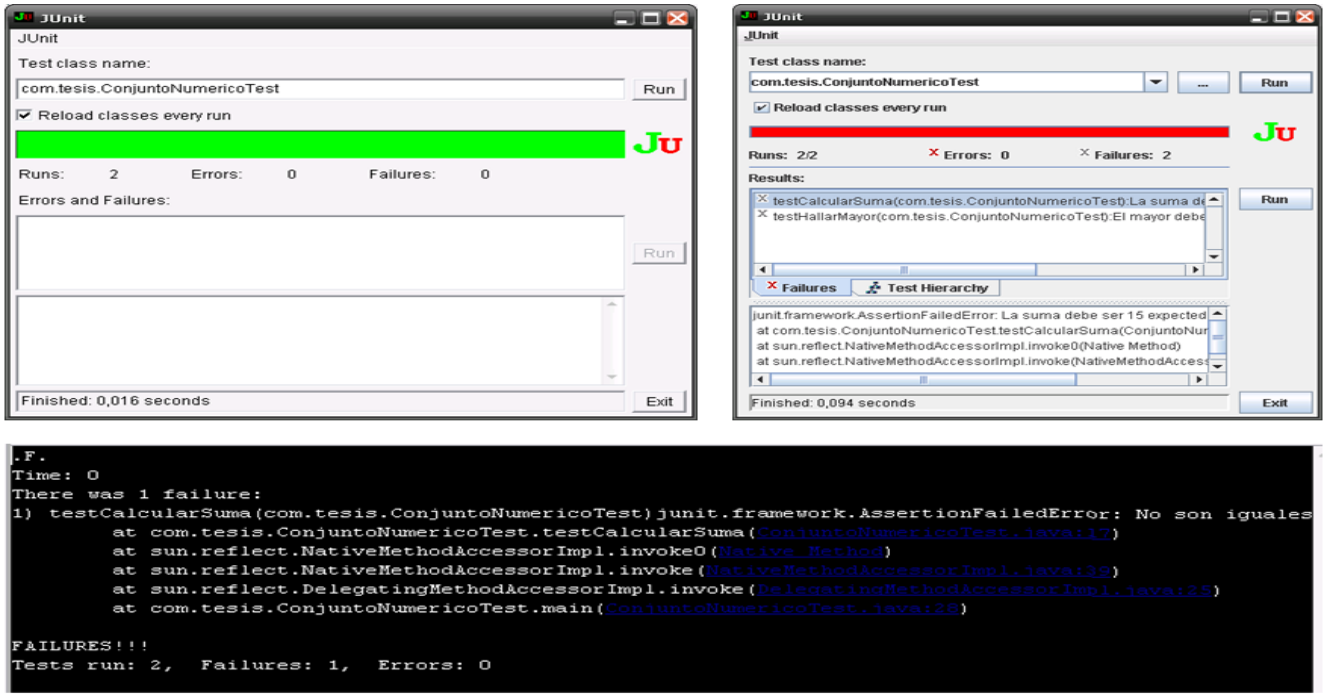

Figura 2.8 Interfaces TestRunner. Las superiores son visuales basadas en SWING y AWT, respectivamente. La inferior es de texto basada en la línea de comandos.

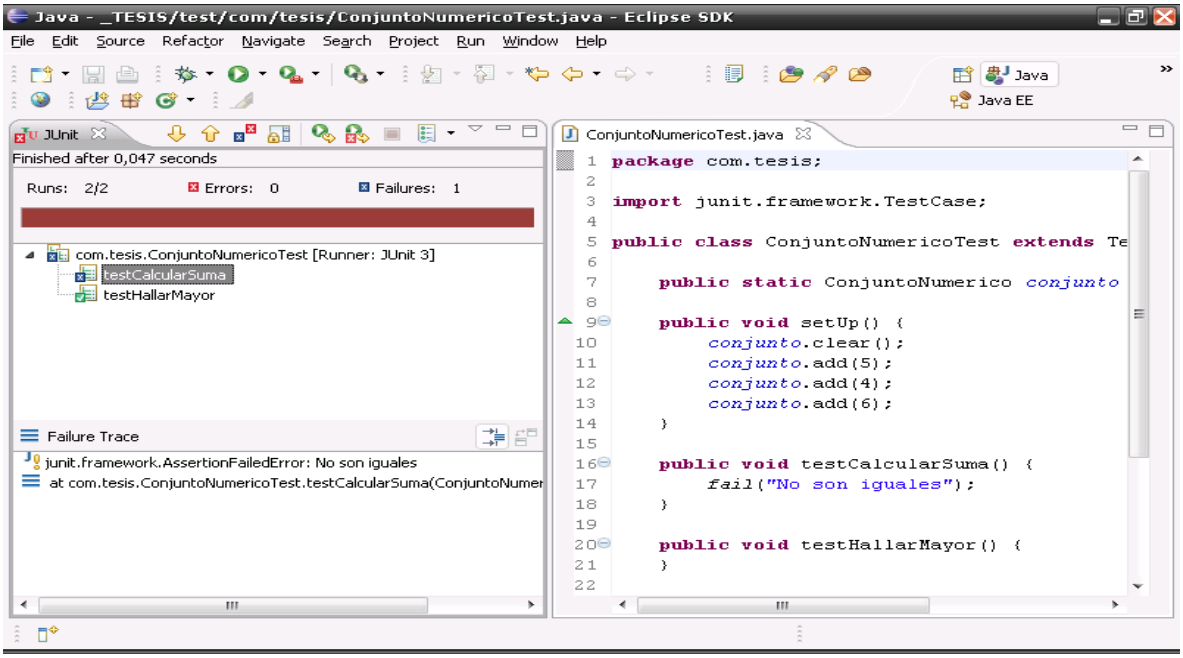

Figura 2.9 JUnit integrado a Eclipse. A la izquierda se muestra una vista basada en TestRunner donde se muestran los resultados de las pruebas realizadas.

También muchos IDEs como Eclipse, traen integrados implementaciones propias de TestRunner para ejecutar pruebas y ofrecer una interfaz gráfica con los resultados obtenidos de su ejecución, según se muestra en la Figura 2.9.

Al ejecutar las pruebas a través de **`TestRunner`**, los desarrolladores se sentirán complacidos si las interfaces visuales muestran la “barra de color verde”. Esto indica que todas las pruebas han sucedido exitosamente. Por el contrario, el color rojo indica que se han detectado fallas en las pruebas, errores que deben ser encontrados y corregidos.

2.2.3 Agentes dobles. Framework MockObject.com.

No siempre es posible realizar pruebas a un componente cuando este depende de otro que no está implementado o no se encuentra disponible en el momento de la prueba. Puede darse el caso también que el componente del cual se dependa ha fallado en las pruebas que se le han aplicado. Sus resultados negativos pueden repercutir en que, las pruebas al primero de estos, también fallen.

Es importante evitar la dependencia entre componentes cuando se realizan las pruebas. Debe lograrse que la porción del sistema a probar sea ejecutada lo más aisladamente posible del resto del sistema. Conocer cómo trabajar con agentes dobles en la realización de las pruebas sería un primer paso en esta tarea. Unas de las técnicas más utilizadas son el uso de las clases de apoyo y de los objetos falsos. Estos reemplazan la porción real del componente del cual el SBP depende de manera parcial o total, según se muestra en la Figura 2.10.

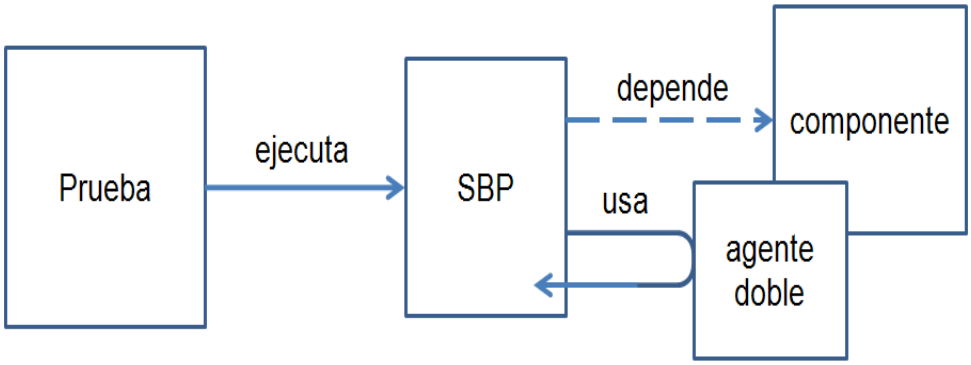

Figura 2.10 La dependencia con otro componente es reemplazada por un agente doble.

Clases de Apoyo.

Las clases de apoyo les permiten a las pruebas tener puntos de control para las entradas indirectas al SBP. Las pruebas podrán forzar a que el SBP se conduzca por caminos que de otra manera no podrían ser ejecutados.

Estas pueden estar dadas por implementaciones a interfaces o ser subclases que redefinan los métodos de alguna clase necesarios para proveer soporte a las funcionalidades requeridas por las pruebas. En algunos casos se convierten en implementaciones temporales hasta que el componente real esté disponible, tal como es mostrado por la Figura 2.11.

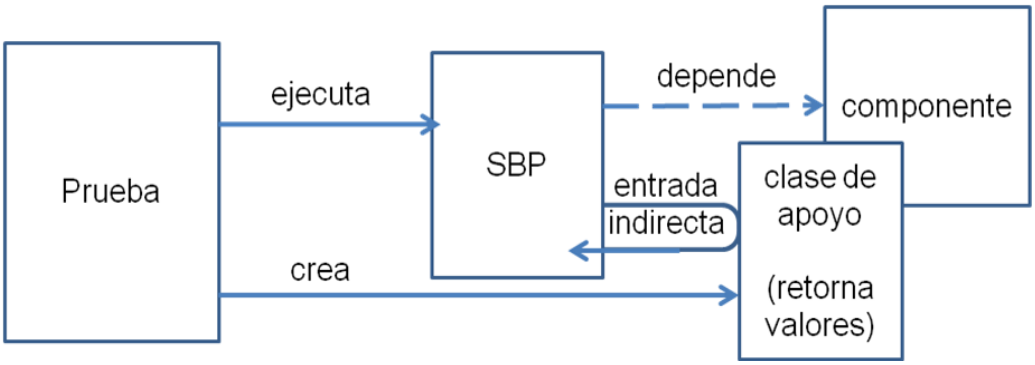

Figura 2.11 Clases de apoyo actuando como agente doble.

***** ***Ejemplo***: Supóngase tener la clase **`PersonaManager`** la cual depende de la interface **`PersonaDAO`** que no cuenta con implementación disponible. Esta clase define un método público donde se debe encontrar el nombre de la persona de mayor edad dada una lista de las personas registradas en el sistema que se obtiene a través de la interface **`PersonaDAO`**. La clase **`Persona`** tiene atributos **`nombre`** y **`edad`** que representan los datos de una persona registrada en el sistema. La interface **`PersonaDAO`** define un método para obtener una lista de personas.

Para hacer una prueba a la clase **`PersonaManager`** que verifique el desempeño correcto de la funcionalidad que retorna el nombre de la persona de mayor edad, se deberá crear una clase de apoyo que implemente la interface **`PersonaDAO`**. El código 2.10 muestra la clase de apoyo **`PersonaDAOStub`**.

Código 2.10 – Clase de apoyo PersonaDAOStub

```
package cu.uci.tesis.capitulo2.stub;
import java.util.List;

public class  implements PersonaDAO{
      private List<Persona> personas;
      public void setPersonas(List<Persona> personas) {
            this.personas = personas;
      }
      public List<Persona> obtenerTodasPersonas() {
```

```
            return personas;
      }
}
```

La clase **PersonaDAOStub** tendrá como atributo una lista de personas, por lo cual no tendrá que conectarse a alguna fuente de datos, ni implementar lógica complicada para obtener la lista requerida. En el código 2.11 se muestra la clase de prueba **PersonaManagerTest**, donde se aprovecharía esta posibilidad.

Código 2.11 – Clase PersonaManagerTest usando la clase de apoyo PersonaDAOStub

```
package cu.uci.tesis.capitulo2.stub;
import java.util.ArrayList;
import java.util.List;
import junit.framework.TestCase;

public class PersonaManagerTest extends TestCase {
      private List<Persona> ;
      public void setUp() {
            List<Persona> personas = new ArrayList<Persona>();
            Persona persona1 = new Persona("Ana", 12);
            Persona persona2 = new Persona("Miguel", 25);
            Persona persona3 = new Persona("Juan", 15);
            personas.add(persona1);
            personas.add(persona2);
            personas.add(persona3);
      }
      public void testNombreMayorEdadDAOStub() {
            PersonaDAOStub personaDAO = new PersonaDAOStub();
            personaDAO.setPersonas();
            PersonaManager manager = new PersonaManager();
            manager.setPersonaDAO(personaDAO);
            String nombre = manager.nombrePersonaMayorEdad();
            assertEquals("nombre debe ser [Miguel]", "Miguel", nombre);
      }
}
```

Esta variante tiene algunos inconvenientes. El primero es que, en caso de que la interface **PersonaDAO** defina más métodos, la clase **PersonaDAOStub** se verá obligada a implementar más métodos, independientemente de que los utilice o no. El segundo de los inconvenientes, es que si existen más dependencias con otros componentes, deben crearse más implementaciones de apoyo, una por cada componente del que se dependa. A esto se une el hecho de que es muy probable que las dependencias cambien constantemente y se deban corregir las pruebas siempre que lo requieran. Esto hace no sólo que el componente a probar dependa de otros componentes sino que también la prueba dependerá de ellos.

Otra variante vendría dada por hacer un buen diseño en la solución, donde la implementación de la funcionalidad a verificar aísle todas las dependencias. Esto significa que no se debe crear dependencias directas con otros componentes sino que se encapsule en otros métodos la necesidad de establecer algún tipo de interacción con ellos.

*** *Ejemplo***: Dado el ejemplo anterior, el cambio de la clase **PersonaManager** quedaría de la manera que se muestra en el código 2.12, donde la lista de personas se recibe a través de un método que decide la manera en que se interactuará con la interface **PersonaDAO**.

Código 2.12 – Clase PersonaManager tiene dependencia con la PersonaDAO

```
package cu.uci.tesis.capitulo2.stub;
import java.util.List;

public class PersonaManager {
      private PersonaDAO ;
      public void setPersonaDAO(PersonaDAO personaDAO) {
            this. = personaDAO;
      }
      public String nombrePersonaMayorEdad( ) {
            List<Persona> personas = this.obtenerPersonas( );
            int mayor = 0;
            for (int i = 0; i < personas.size( ); i++) {
                  if (personas.get(i).getEdad( ) > personas.
                                                      get(mayor).getEdad( ))
                        mayor = i;
            }
            return personas.get(mayor).getNombre();
```

```
      }
      protected List<Persona> obtenerPersonas(){
            return.obtenerTodasPersonas();
      }
}
```

En el código 2.13 muestra cómo se crearía una subclase de **`PersonaManager`**, donde la esencia de la solución estaría dada por redefinir la funcionalidad de la cual se depende en lugar del supuesto componente que no está disponible.

Código 2.13 – Clase PersonaManagerStub donde se implementa la funcionalidad requerida

```
package cu.uci.tesis.capitulo2.stub;
import java.util.List;

public class PersonaManagerStub extends PersonaManager {
      private List<Persona> ;
      public void setPersonas(List<Persona> personas) {
            this. = personas;
      }
      protected List<Persona> obtenerPersonas() {
            return ;
      }
}
```

En el código 2.14 se muestra cómo quedaría entonces la implementación de la clase **`PersonaManagerTest`** usando la clase de apoyo **`PersonaManagerStub`**.

Código 2.14 – Clase PersonaManagerStub donde se implementa la funcionalidad requerida

```
package cu.uci.tesis.capitulo2.stub;
import java.util.ArrayList;
import java.util.List;
import junit.framework.TestCase;

public class PersonaManagerTest extends TestCase {
      private List<Persona> personas;
```

```
	public void setUp() {
		personas = new ArrayList<Persona>();
		Persona persona1 = new Persona("Ana", 12);
		Persona persona2 = new Persona("Miguel", 25);
		Persona persona3 = new Persona("Juan", 15);
		personas.add(persona1);
		personas.add(persona2);
		personas.add(persona3);
	}
	public void testNombreMayorEdadManagerStub() {
		PersonaManager manager = new PersonaManagerStub();
		((PersonaManagerStub)manager).setPersonas(personas);
		String nombre = manager.nombrePersonaMayorEdad();
		assertEquals("nombre debe ser [Miguel]", "Miguel", nombre);
	}
}
```

Esta manera de realizar las pruebas será válida siempre que no se modifique el verdadero propósito a probar. A diferencia de la variante anterior, en este caso la prueba no dependerá de otras interfaces, se implementan sólo los métodos necesarios para satisfacerla y se necesitará mantenimiento sólo cuando los cambios en la funcionalidad a probar repercutan en la prueba. Se obtiene una prueba más robusta pero implica que se debe tener en cuenta la necesidad de la prueba antes de diseñar e implementar lo que será probado.

Crear clases de apoyo en muchas ocasiones es difícil aunque se puedan reescribir selectivamente los métodos necesarios. Esta forma no es buena cuando crear la clase introduce dependencias con objetos que son irrelevantes y se requiera de mucho esfuerzo para crearlos. Si un componente depende de otros componentes, deberá preguntarse si es conveniente reescribir todos y cada uno de los comportamientos esperados o crear nuevas clases de apoyo para cada método de prueba. También puede ser problemático el hecho de que se tenga que simular un complejo ambiente, tal como el contexto de un contenedor web.

Objetos falsos.

Los objetos falsos son usados para implementar verificación de comportamiento y evitar la duplicación de código entre pruebas similares. En la figura 2.12 se muestra cómo se delega el trabajo de verificar las salidas indirectas del SBP completamente al agente doble.

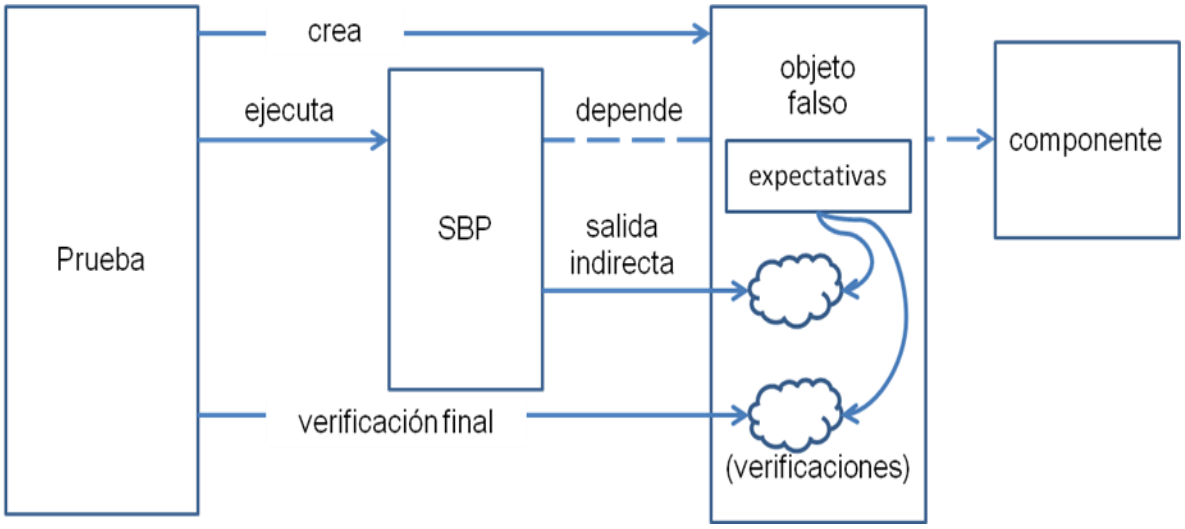

Figura 2.12 Objetos falsos actuando como agentes dobles

Un objeto falso se crea especificando las mismas interfaces que implementa el componente del cual depende el SBP. En la prueba se establecen las expectativas a cumplir por parte de este objeto falso, que no son más que las llamadas a métodos que debe esperar del SBP e indicando los valores con los cuales debe responder al mismo. Al ejecutar el SBP se sitúa el objeto falso en lugar del componente del que se depende. El objeto falso comparará los argumentos recibidos en cada llamada con los argumentos esperados usando aserciones de igualdad y fallará si no coinciden. Al finalizar la prueba verificará que todas las expectativas hayan sido cumplidas. Así la prueba no debe preocuparse siempre por crear y verificar las aserciones establecidas.

Existen varios frameworks basados en JUnit que implementan estas ideas. Algunos son **`MockObject.com`**, **`EasyMock`**, **`jMock`** y **`MockMaker`**.

***** ***Ejemplo***: Se tiene una clase **`ArticuloManager`** que gestiona la compra de artículos cuya información se encuentra en alguna fuente de datos a la cual se accede a través de la interface **`ArticuloDAO`**. Dicha interfaz define dos métodos: uno para obtener un objeto representando todos los datos de un artículo conocido su código; el otro para actualizar cualquier cambio que se haya efectuado en los datos de un articulo dado. La clase **`Articulo`** contiene los atributos **`codigo`**, **`valor`** y **`existencia`** que indican el código de un articulo, el precio que tiene por unidad y la cantidad en existencia, respectivamente. Es necesario redefinir la implementación del método **`equals()`** en la clase **`Articulo`** debido a que los objetos falsos hacen verificaciones de igualdad entre objetos a través de dicho método para saber si son cumplidas sus expectativas. El código 2.15 muestra cómo se realizaría la prueba correspondiente, utilizando objetos falsos, donde no se necesitaría implementar otras clases de apoyo.

Código 2.15 – Clase ArticuloManagerTest usando Objetos Falsos

```
package cu.uci.tesis.capitulo2.mock;
import junit.framework.TestCase;
import com.mockobjects.dynamic.Mock;

public class ArticuloManagerTest extends TestCase {
      private Mock mockDAO;
      private ArticuloManager articuloManager;
      public void setUp() {
            mockDAO = new Mock(ArticuloDAO.class);
            Articulo articulo = new Articulo("COD-001", 10.0f, 50);
            mockDAO.expectAndReturn("obtenerArticuloPorCodigo",
                                                    "COD-001", articulo);
            articuloManager = new ArticuloManager();
            articuloManager.setArticuloDAO((ArticuloDAO)mockDAO.proxy());
      }
      public void testEfectuarCompraObtenerPrecio() {
            Articulo articuloActualizado = new Articulo("COD-001",
                                                                  10.0f, 30);
            mockDAO.expect("actualizarArticulo", articuloActualizado);
            try {
                  float precio = articuloManager.
                                    efectuarCompraObtenerPrecio("COD-001",20);
                  assertEquals("El precio a cobrar debe ser 20*10=200",
                                                          200.0f, precio);
            } catch (Exception error) {
                  fail(error.getMessage());
            }
      }
      public void testEfectuarCompraPorEncimaExistencia() {
            try {
                  articuloManager.
                                    efectuarCompraObtenerPrecio("COD-001", 70);
                  fail("Se espera un fallo");
            } catch (Exception error) {
                  assertTrue(error.getMessage(), true);
```

```
            }
        }
}
```

En el método **setUp()** se crea el objeto falso (**mockDAO**). Se especifica cuál será el nombre del método que esperará ser llamado, cuál será el parámetro y con qué valor debe responder a través del método **expectAndReturn()**. Luego se inyecta en el objeto de la clase **ArticuloManager** que será objetivo de la prueba. Cada método de prueba interactuará con los métodos de la clase **ArticuloManager** y esta, a su vez, con los métodos de la interface **ArticuloDAO** a través del objeto falso. Por su parte, cada método de prueba puede indicar otras expectativas a cumplir por el objeto falso.

En los dos métodos de pruebas presentados se persiguen objetivos distintos. En el primero, se indican entradas directas e indirectas que buscan el funcionamiento exitoso de la funcionalidad. En el segundo, se induce a un estado en el que se debe lanzar una excepción para ver si el SBP es capaz de comportarse de la manera esperada y puede lidiar con el problema.

Los objetos falsos son más efectivos si se sigue buenas prácticas de programación a través de interfaces y se tiene un conjunto manejable de responsabilidades en cada clase. Los objetos falsos no son la solución a todo tipo de problemas de este tipo. Generalmente, son usados para pruebas de caja blanca lo cual puede provocar que las pruebas fallen si cambian las implementaciones de los componentes. Las clases de apoyo en ocasiones son más apropiadas.

Hasta este momento se han identificado las principales maneras de escribir las pruebas unitarias para funcionalidades simples. En próximos epígrafes se abordarán las pruebas a componentes con mayor complejidad, tanto por su estructura como por el marco en que deben ser ejecutadas.

2.2.4 Framework DbUnit.

Muchas aplicaciones usan una base de datos para almacenar el estado persistente de la aplicación por lo que, en ocasiones, algunas pruebas requerirán acceder a la base de datos. Una base de datos es una causa primaria de pruebas erradas debido a que se persisten muchos datos entre prueba y prueba. Esto es especialmente difícil de evitar cuando el entorno de desarrollo contiene una sola base de datos y todas las pruebas ejecutadas por los desarrolladores interactúan con esta. Sería recomendable separar los datos a los que acceden pruebas distintas. Una manera puede ser teniéndose varias instancias de la misma base de datos, una para cada desarrollador que ejecute pruebas, para evitar conflictos en la concurrencia.

Otra cuestión sería el hecho de que el probador debe crear configuraciones conocidas en todo momento por las pruebas. Existen muchos modos de lograr una configuración estable para una suite de pruebas. Se puede tener la base de datos en un estado a partir del cual serán ejecutadas las pruebas o bien se pueden realizar un conjunto de acciones, puede ser a través de un script de sentencias SQL, que establezcan el estado necesitado por las pruebas antes de ser ejecutadas.

En el caso de las pruebas unitarias con código persistente no es factible tener agentes dobles. Se convierte en una labor muy trabajosa simular sentencias, procedimientos almacenados, restricciones de integridad referencial, conexiones, etcétera, por lo que no es común aislar los componentes de acceso a datos. Entre los frameworks xUnit para realizar pruebas al código de acceso a datos para aplicaciones en Java, está DbUnit.

DbUnit es una extensión de JUnit para proyectos que utilizan bases de datos. Entre otras cosas, su objetivo es colocar la base de datos en un estado conocido para la ejecución de las pruebas. Ayuda a evitar muchos problemas que usualmente ocurren cuando una prueba corrompe la base de datos y subsiguientes pruebas puedan fallar o empeorar el daño. DbUnit tiene la habilidad de exportar e importar los datos de la base de datos hacia y desde ficheros XML y puede verificar si la información contenida en la base de datos coincide con conjuntos de valores esperados.

Para realizar pruebas con DbUnit se crea una subclase de **`DatabaseTestCase`**. Se deben implementar los métodos **`getConnection()`** y **`getDataSet()`** que proveerán la conexión a la base de datos y los datos que debe contener la misma antes de ejecutar los métodos de prueba, respectivamente. En los métodos de prueba se puede comparar el estado actual de la base de datos con datos esperados obtenidos desde un fichero XML.

***** ***Ejemplo***: Supóngase tener una clase **`PersonaDAO`**, con un método **`insertarPersona()`** que recibe por parámetro los datos de una persona para ser almacenados en la base de datos. La clase **`Persona`** tiene atributos **`nombre`** y **`edad`**, representando los datos de la tabla **`PERSONA`** en la base de datos con los campos **`NOMBRE`** y **`EDAD`**, respectivamente. Para ejecutar la prueba se definen los datos requeridos en el fichero **`testData.xml`** con el formato requerido por DbUnit, que representan los datos que serán insertados en la base de datos, para establecer el estado necesario, según se muestra en el código de ejemplo 2.16.

Código 2.16 – fichero testData.xml que contiene los datos necesarios para ejecutar la prueba

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
    <PERSONA nombre="Ana" edad="12"/>
```

```
    <PERSONA nombre="Miguel" edad="25"/>
    <PERSONA nombre="Juan" edad="18"/>
</dataset>
```

La etiqueta **<dataset>** encierra la declaración de todos los datos a definir. La etiqueta **<PERSONA>**, en este caso indica el nombre de la tabla en la que se insertarán tales datos. Las propiedades **nombre** y **edad** indican el nombre de las columnas y los valores de cada propiedad indican los valores que tendrán respectivamente los campos de cada record en la base de datos. Luego de esto, la clase de prueba quedaría según se muestra en el código 2.17 usando DbUnit.

Código 2.17 – clase PersonaDAOTest usando DbUnit

```
package cu.uci.tesis.capitulo2.dbunit;
import java.io.FileInputStream;
import org.apache.commons.dbcp.BasicDataSource;
import org.dbunit.Assertion;
import org.dbunit.DatabaseTestCase;
import org.dbunit.database.DatabaseDataSourceConnection;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.ITable;
import ;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.dbunit.operation.DatabaseOperation;

public class PersonaDAOTest extends DatabaseTestCase {
     private PersonaDAO personaDAO;;
     public PersonaDAOTest(String name) {
          super(name);
     }
     protected DatabaseOperation getSetUpOperation() throws Exception {
          this.personaDAO = new PersonaDAO();
          return DatabaseOperation.CLEAN_INSERT;
     }
     protected DatabaseOperation getTearDownOperation()throws Exception{
          return DatabaseOperation.NONE;
     }
```

```
	public void testInsertarPersona() {
		Persona persona = new Persona("Roberto", 47);
		try {
			personaDAO.insertarPersona(persona);
			ITable actualTable = getConnection().createDataSet().
                                                    getTable("PERSONA");
			IDataSet expectedDataSet = new FlatXmlDataSet(
                                new FileInputStream( "expectedData.xml"));
			ITable expectedTable = expectedDataSet.
                                                    getTable("PERSONA");
			Assertion.assertEquals(new (expectedTable),
				new (actualTable, expectedTable.
                                                  getTableMetaData()));
		} catch (Exception error) {
			fail(error.getMessage());
		}
	}
	protected IDatabaseConnection getConnection() throws Exception {
		BasicDataSource dataSource = new BasicDataSource();
		dataSource.setUrl("jdbc:postgresql://localhost/tesis");
		dataSource.setDriverClassName("org.postgresql.Driver");
		dataSource.setUsername("postgres");
		dataSource.setPassword("postgres");
		IDatabaseConnection connection = new
                            DatabaseDataSourceConnection(dataSource);
		return connection;
	}
	protected IDataSet getDataSet() throws Exception {
		return new FlatXmlDataSet(
                                new FileInputStream("testData.xml"));
	}
}
```

En el método **`getConnection()`** se obtiene la configuración necesaria para establecer la conexión con la base de datos. En el método **`getDataSet()`** se obtienen los datos requeridos para ejecutar la prueba a través de la configuración de los mismos en un fichero XML. En los métodos

`getSetUpOperation()` y **`getTearDownOperation()`** se definen las operaciones a ejecutar con los datos en la base de datos antes y después de la ejecución de la prueba, respectivamente. En el método de prueba **`testInsertarPersona()`**, luego de ejecutar la operación de prueba del objeto de la clase **`PersonaDAO`**, se obtiene una descripción de los datos esperados desde otro fichero XML y estos son comparados con los datos actuales en la base de datos.

El contenido del fichero **`expectedData.xml`** con la descripción del estado esperado en la base de datos se muestra en el código 2.18:

Código 2.18 – fichero expectedData.xml que contiene los datos a ser verificados

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
      <PERSONA nombre="Ana" edad="12" />
      <PERSONA nombre="Miguel" edad="25" />
      <PERSONA nombre="Juan" edad="18" />
      <PERSONA nombre="Roberto" edad="47" />
</dataset>
```

2.2.5 Frameworks ServletUnit y HttpUnit.

Es más difícil realizar pruebas a componentes web que se desarrollen como parte de la aplicación. En estos casos, es necesario acceder a los elementos web sin tener que navegar por un camino que puede ser largo o difícil de transitar, según se muestra en la Figura 2.13.

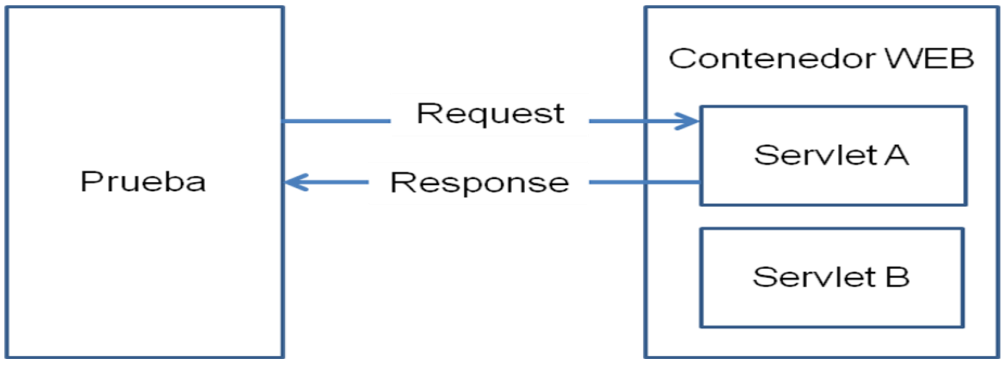

Figura 2.13 Las pruebas deben simular el comportamiento en un contenedor WEB

HttpUnit y ServletUnit se integran en función de hacer que esto sea sencillo. HttpUnit emula las porciones relevantes del comportamiento del browser incluyendo envío de formularios, código JavaScript, cookies, etcétera, y permite que el código de prueba en Java examine el contenido de las

páginas retornadas tales como textos, XML DOM, contenedores de formularios, tablas y links. La misma técnica es usada por ServletUnit, el cual posibilita simular contenedores web.

Una clase de prueba basada en ServletUnit y HttpUnit se realizará como una subclase de **`ServletTestCase`**, donde se registran los **`Servlets`** (instancias de las clases a ser probadas) que se utilizarán en la ejecución de la prueba. En los métodos de prueba se podrán obtener instancias de **`WebRequest`** y **`WebResponse`** que representan las entradas y las salidas en la ejecución de las peticiones a ser probadas. Se podrán crear verificaciones de aserción a través de las propiedades obtenidas de objetos de clases tales como **`WebTable`**, **`WebForm`**, entre otros.

***** ***Ejemplo***: La clase **`PLanguageServlet`**, la cual representa un **`Servlet`**, procesará una petición tal que reciba como parámetro el lenguaje de programación favorito del que envía la petición. La respuesta del mismo será un texto formato HTML conteniendo una tabla con los nombres de los posibles libros recomendados para dicho lenguaje, según se muestra en la figura 2.14 para el caso en que el lenguaje seleccionado sea **`Java`**.

Figura 2.14 Respuesta esperada por la petición.

La clase de prueba implementada para este caso quedaría según se muestra en el código 2.19.

Código 2.19 – clase PLanguageServletHttpUnitTest usando ServletUnit y HttpUnit

```
package cu.uci.tesis.capitulo2.web;
import com.meterware.httpunit.PostMethodWebRequest;
import com.meterware.httpunit.WebRequest;
import com.meterware.httpunit.WebResponse;
import com.meterware.httpunit.WebTable;
import com.meterware.servletunit.ServletRunner;
```

```
import com.meterware.servletunit.ServletTestCase;
import com.meterware.servletunit.ServletUnitClient;

public class PLanguageServletHttpUnitTest extends ServletTestCase {
      private ServletRunner servletRunner;
      public ProgrammingLanguageServletHttpUnitTest(String name) {
            super(name);
      }
      public void setUp() throws Exception {
            super.setUp();
            servletRunner = new ServletRunner();
            servletRunner.registerServlet("servlet.htm",
                        ProgrammingLanguageServlet.class.getName());
      }
      public void testPostMethod() throws Exception {
            ServletUnitClient servletClient = servletRunner.newClient();
            WebRequest request = new PostMethodWebRequest(
                        "http://test.meterware.com/servlet.htm");
            request.setParameter("lenguaje", "Java");
            WebResponse response = servletClient.getResponse(request);
            assertNotNull("Respuesta recibida", response);
            assertEquals("ContentType", "text/html",
                                                response.getContentType());
            assertEquals("Titulo", "Lenguaje Preferido",
                                                      response.getTitle());
            WebTable table = response.getTables()[0];
            assertEquals("filas", 2, table.getRowCount());
            assertEquals("columnas", 2, table.getColumnCount());
      }
}
```

En el método **`setUp()`** se instancia un **`ServletRunner`** que simulará el funcionamiento de un contenedor web. Dentro del mismo se registra el **`Servlet`** a probar con la dirección "**`/servlet.htm`**". En el método de prueba **`testPostMethod()`** se instancia un cliente que ejecutará la petición y se obtiene un **`WebRequest`** al cual se le establecen parámetros. De este último

se obtiene un **WebResponse** conteniendo la información de la respuesta a verificar a través de las propiedades de objetos de tipo **WebTable**, entre otros.

2.2.6 JUnit4 para pruebas de Nueva Generación.

JUnit4 usa anotaciones[35] de Java para identificar cuáles son los métodos de prueba. Para versiones anteriores, estos métodos de pruebas eran identificados por técnicas de reflexión[36] aplicadas a los métodos de la clase de prueba, los cuales debían comenzar su nombre con **test**. Todo esto es posible a partir de la versión 5.0 de Java, a partir de la cual se han introducido muchas de estas bondades del lenguaje.

Clases y métodos de prueba.

Para escribir una prueba con JUnit4 basta con crear una clase (que no está obligada a ser subclase de ninguna otra) y anotar un método con **@Test**. En el código 2.20 se muestra una clase de prueba con un método de prueba utilizando JUnit4.

Código 2.20 – Clase de prueba con un método de prueba anotado con @org.junit.Test

```
package cu.uci.tesis.capitulo3.junit4;
import org.junit.Test;

public class ClassJUnit4Test {
      @Test
      public void metodoDePrueba() {
            // ....
      }
}
```

Esto permite que los métodos de pruebas puedan tener cualquier nombre, lo cual puede ser favorable, para quienes sean escépticos en nombrar los métodos de prueba de alguna manera rígidamente

[35] Las anotaciones de Java (Java Annotations en inglés) son etiquetas establecidas, que proveen de información acerca de un programa. No tienen efecto directo en las operaciones definidas en el código. Pueden ser anotados clases, métodos, variables, parámetros y paquetes.

[36] Las reflexiones (Java Reflection en inglés) son técnicas del lenguaje a través de las cuales se accede a los metadatos de un programa (por ejemplo, a las propiedades de una clase o interface, tales como los métodos y atributos declarados, parámetros, etcétera.)

establecida. Además, el que la clase de prueba no esté obligada a ser subclase de alguna clase o interface definida por el framework JUnit, permite que se puedan heredar funcionalidades de otras clases en algún momento si es que se desea.

Configuración.

En versiones anteriores de JUnit, se definían métodos **`setUp()`** y **`tearDown()`** para establecer y desmontar configuraciones correspondientes a conjuntos de datos que eran compartidos por más de un método de prueba. JUnit4 brinda una forma de anotar métodos declarados en la clase de prueba para lograr el mismo propósito con ellos. Cuando se anota un método con **`@Before`** y otro método con **`@After`**, estos se ejecutarán antes y después de cada instancia de prueba, similar a los métodos **`setUp()`** y **`tearDown(),`** al ser ejecutada la suite de pruebas, respectivamente. En el código 2.21 se muestra un ejemplo de esto.

Código 2.21 – Clase de prueba con anotaciones @org.junit.Before y @org.junit.After

```
package cu.uci.tesis.capitulo3.junit4;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class ClassJUnit4Test {
	@Before
	public void iniciandoPrueba() {
		// ....
	}
	@Test
	public void metodoDePrueba() {
		// ....
	}
	@After
	public void () {
		// ....
	}
}
```

El método que sea anotado con **@Before** se encargará de preparar el entorno donde será ejecutado el SBP y el método anotado con **@After** lidiará con desmontar cualquier detalle de la configuración anterior que pueda causar problemas en otras pruebas.

Aserciones y excepciones.

Como estas clase de prueba no heredan de las clases convencionales de pruebas brindadas por el framework JUnit, no disponen de los métodos **assertXXX()** como parte de sus funcionalidades implícitas. Cuando se quieran verificar algunos valores, se pueden importar estáticamente los métodos de **org.junit.Assert.*** y ser usados como los conocidos métodos **assertXXX()**, según se muestra en el ejemplo del código 2.22.

Código 2.22 – Clase de prueba importando estáticamente org.junit.Assert.*

```
package cu.uci.tesis.capitulo3.junit4;
import org.junit.Test;
import static org.junit.Assert.*;

public class ClassJUnit4Test {
      @Test
      public void metodoDePrueba() {
            // ....
            assertEquals("Mensaje","esperado","actual");
      }
}
```

La sentencia "**import static org.junit.Assert.***" permite utilizar todo el contenido de tal paquete como si fuese parte de la clase que lo importa.

Por otro lado, en un método de prueba se puede indicar el tipo de excepción que se espera cuando es posible que alguna sentencia aborte debido al lanzamiento de un error determinado. La anotación **@Test** tiene un parámetro opcional llamado **expected** que toma valores de subclases de **Throwable**, según se muestra en el ejemplo del código 2.23.

Código 2.23 – Clase de prueba donde se esperan excepciones.

```
package cu.uci.tesis.capitulo3.junit4;
import org.junit.Test;
```

```
import static org.junit.Assert.*;

public class ClassJUnit4Test {
      @Test(expected = ArithmeticException.class)
      public void metodoDePrueba() {
            float division = 1 / 0;
            fail("División no permitida");
      }
}
```

De esta manera, al ser ejecutado el método de prueba, se verificará que sea lanzada una excepción que sea subclase de la señalada por la anotación correspondiente. Este método también fallará si no es lanzada ninguna excepción.

Suites de pruebas.

Para que estas implementaciones sean ejecutadas en una suite de pruebas y se recolecten los resultados, JUnit4 define una manera de que puedan acceder al TestRunner diseñado para ejecutar las pruebas. Para ello similar a las suites de pruebas en versiones anteriores se debe crear un método estático **`suite()`** según muestra el código 2.24.

Código 2.24 – Clase para crear una suite de prueba.

```
package cu.uci.tesis.capitulo3.junit4;
import junit.framework.JUnit4TestAdapter;
import junit.framework.Test;

public class ClassJUnit4TestSuite {
      public static Test suite() {
            return new JUnit4TestAdapter(ClassJUnit4Test.class);
      }
}
```

De esta manera se convierte la clase de prueba implementada basada en JUnit4 en una instancia equivalente a versiones anteriores, que puede ser ejecutada luego por el **`TestRunner`** deseado. Otra manera de crear suites de pruebas, es utilizando precisamente las anotaciones que brinda JUnit4,

donde se indican cuales clases de prueba serán parte de la suite de una clase determinada. En el código 2.25 se muestra un ejemplo de ello.

Código 2.25 – Clase para crear una suite con varias clases de prueba

```
package cu.uci.tesis.capitulo3.junit4;
import junit.framework.JUnit4TestAdapter;
import junit.framework.Test;
import org.junit.runners.Suite.SuiteClasses;

@SuiteClasses( { ClassJUnit4Test.class, ClassJUnit4TestSuite.class })
public class ClassJUnit4TestAll {
      public static Test suite() {
            return new JUnit4TestAdapter(ClassJUnit4TestAll.class);
      }
}
```

Con la anotación **`@SuiteClasses`** se indican en un arreglo de clases las que formarán parte de la suite a definir en la clase correspondiente.

2.3 Herramientas de análisis de cobertura de código.

La mayoría de los propósitos de las pruebas serán cumplidos si tienen total cobertura del código de producción o, al menos, cubren una parte significativa. Una porción de código que no ha sido probada es una parte no confiable del sistema y un lugar donde es posible encontrar errores en algún momento. Tener código que no ha sido ejercitado durante alguna prueba, cuando se han cubierto todos los escenarios posibles, indica que se tiene código no accesible, que nunca será ejecutado, por lo que se puede prescindir de él.

Saber si el código está siendo probado a fondo, sin asumirlo ni tomar suposiciones estimadas, ayudaría a tomar decisiones de qué pruebas se necesitarían implementar. De esta manera se podría conocer si las pruebas existentes cumplen las condiciones necesarias para obtener lo que se espera de ellas.

Existen muchas herramientas para tracear el código que es ejercitado durante la ejecución de pruebas con JUnit. Muchos de estos se integran a IDEs como Eclipse y pueden producir reportes de los resultados obtenidos en varios formatos. De manera general, muestran porcientos del código que es

ejecutado respecto al total y cuáles son las sentencias del código que han sido o no ejecutadas. Dos de estas herramientas son EclEmma y CodeCover.

2.3.1 EclEmma.

EclEmma es una herramienta libre, galardonada en los “Premios de la Comunidad de Eclipse 2008” como “Mejor Herramienta de Desarrollo de Código Abierto basada en Eclipse”. Es empleada para determinar la cobertura de código Java. Se ejecuta dentro de área de trabajo similar a la ejecución de pruebas JUnit, desde donde puede directamente analizar la cobertura de código. Los resultados que obtiene son inmediatamente resumidos y mostrados en editores de código Java. No requiere modificar la estructura de los proyectos ni hacer otras configuraciones en los mismos. Su principal objetivo en la integración con Eclipse es mantener estrecha interactividad con el desarrollador al ejecutar las pruebas, según se muestra en la Figura 2.15.

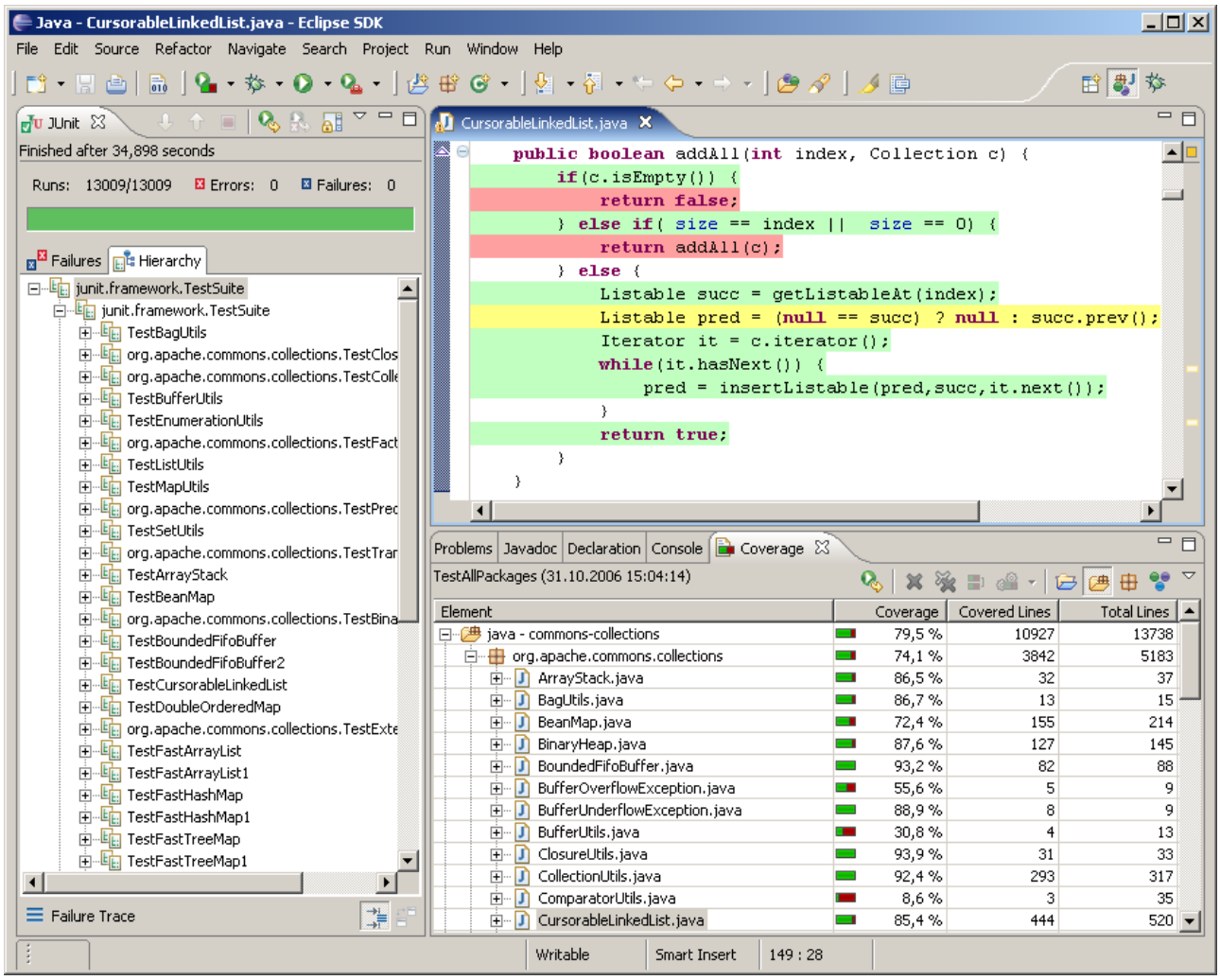

Figura 2.15 Vistas de las funcionalidades de EclEmma dentro de Eclipse

EclEmma añade un modo de ejecución al área de trabajo de Eclipse (**`Coverage`**), que funciona similar a los de ejecución (**`Run`**) y depuración (**`Debug`**). Este puede ser activado desde el menú de ejecución o desde la barra de menú, similar al mostrado en la Figura 2.16.

Figura 2.16 Barra de menú insertada por EclEmma en Eclipse

Cuando son ejecutadas las pruebas se dispone automáticamente de la información de cobertura del código. En la vista de cobertura (**`Coverage View`**) se obtiene una lista resumida de cobertura del código para los proyectos Java. Donde se muestra el número total de elementos y cuántos de estos han sido cubiertos. Se incluye la jerarquía respectiva por paquetes, clases, métodos, etcétera. Similar al mostrado en la Figura 2.17.

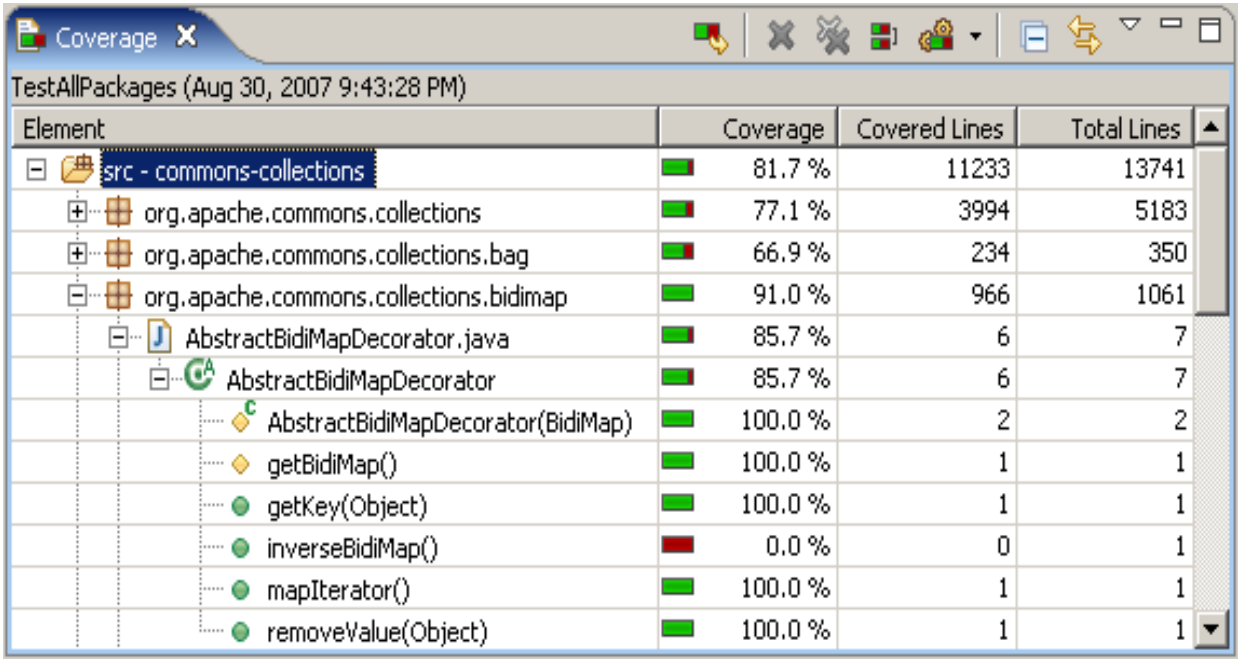

Figura 2.17 Vista del análisis de cobertura mostrado por EclEmma.

La cobertura es directamente visible en los editores de código de Java. Con colores personalizados se destacan las sentencias de código que han sido cubiertas o no de manera total o parcialmente. En la Figura 2.18 se muestra la manera estándar en que las herramientas de este tipo definen códigos de colores que indican el grado de cobertura en las sentencias: el color verde indica que la línea ha sido totalmente cubierta, le amarillo para las líneas que han sido cubiertas parcialmente y el rojo para las líneas que no fueron ejecutadas.

```
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index ||  size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

Figura 2.18 Código Java destacado con colores que indican el grado de cobertura.

EclEmma no solo está diseñado para ejecutar pruebas y analizar cobertura de código, sino que también provee funciones para importar o exportar resultados. Se puede importar ficheros ****.ec*** que contienen datos de resultados de otros análisis de cobertura y se puede exportar no solo a ficheros ****.ec*** sino también a ficheros en formato XML o HTML, tal como muestra la Figura 2.19.

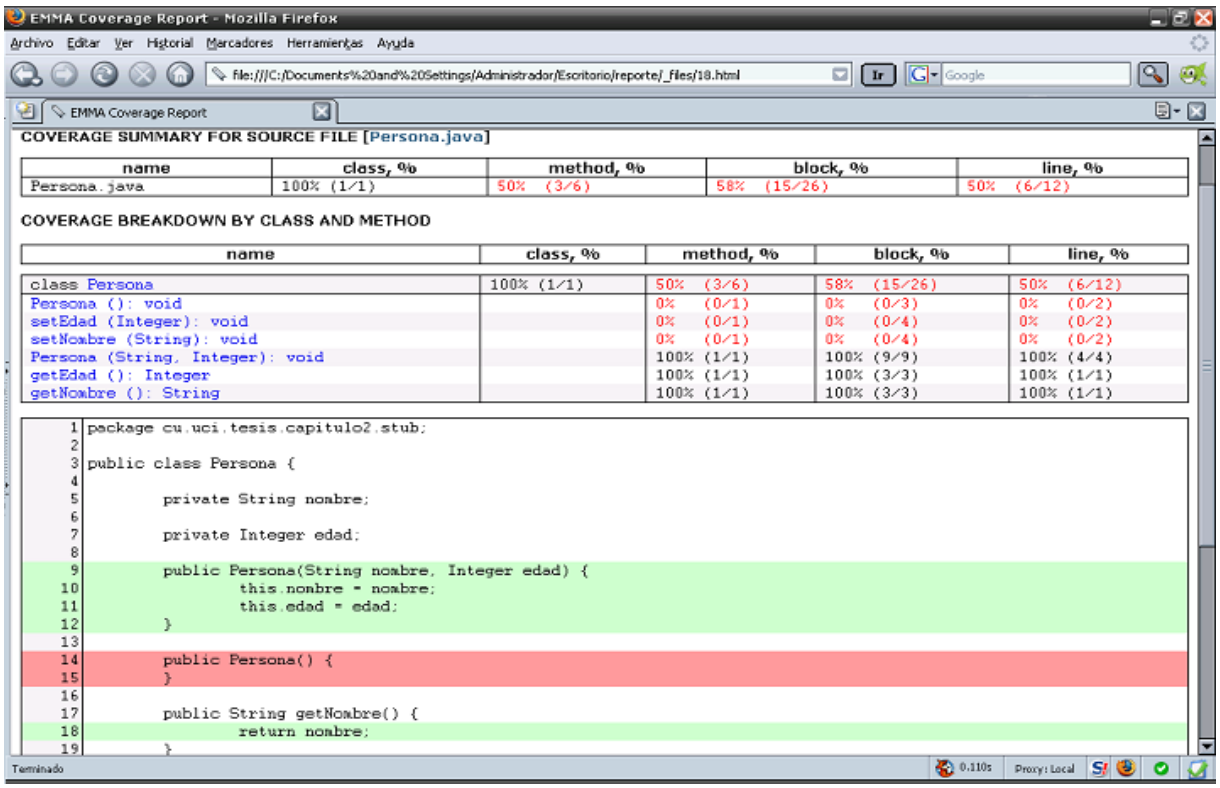

Figura 2.19 Reporte en formato HTML con los resultados de cobertura, obtenidos y exportados por EclEmma.

De esta manera se tiene constancia histórica de los resultados de las pruebas. Permite hacer evaluaciones con otros resultados de pruebas para medir el comportamiento que han tenido los resultados a lo largo de la etapa de implementación del código de producción.

2.3.2 CodeCover.

CodeCover es una herramienta libre de pruebas de caja negra, desarrollada en la Universidad de Stuttgart en el año 2007. Aplica medidas a sentencias, bifurcaciones, ciclos, entre otros tipos de cobertura. CodeCover presenta interesantes vistas cuando se integra con Eclipse, las que pueden verse en la Figura 2.20. Algunas son: la matriz de correlación de pruebas que muestra cuán similar pueden ser dos pruebas entre sí evitando que ambas prueben lo mismo, el analizador binario (boolean) que ayuda a entender la cobertura de las condiciones y la vista de cobertura que filtra las clases y métodos que tienen una cobertura menor o mayor a un valor seleccionado.

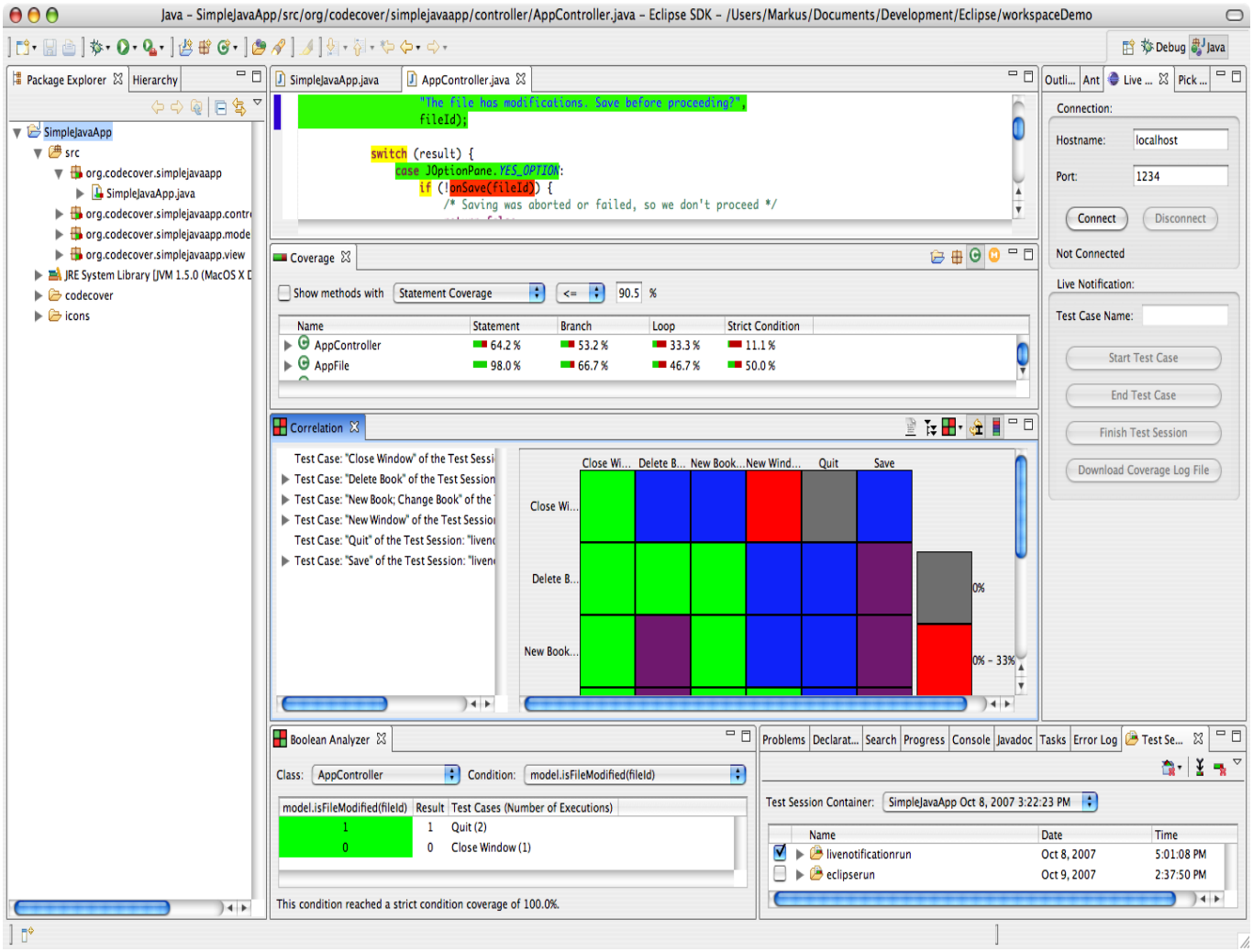

Figura 2.20 Vistas de las funcionalidades que ofrece CodeCover

Para habilitar CodeCover en un proyecto al cual se le aplicará medidas de cobertura, se deben editar sus propiedades en la categoría de CodeCover. En la parte izquierda de la Figura 2.21 se muestra dónde se seleccionan los criterios de análisis de cobertura que se deseen usar. En la parte derecha de la Figura 2.21 se muestra la vista del explorador de paquetes donde se eligen las clases a las cuales se les aplicarán los análisis de cobertura, a través del menú de contexto (clic derecho) e indicando la opción **`Use For Coverage Measurement`**.

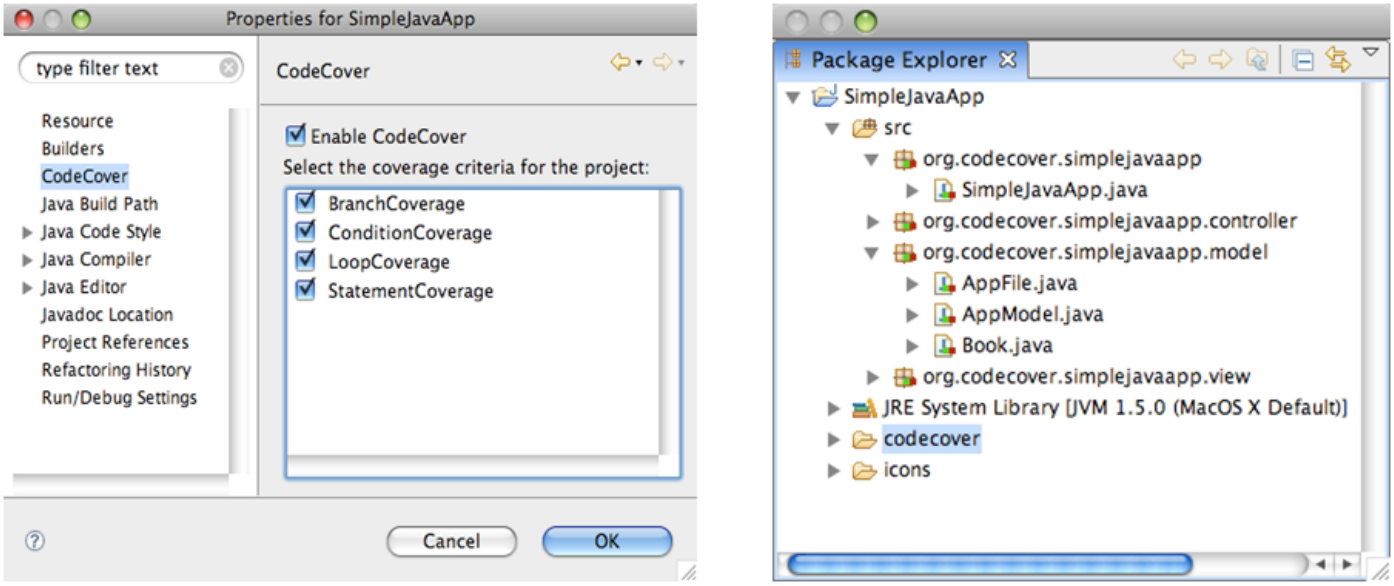

Figura 2.21 Editando propiedades para establecer cobertura con CodeCover en un proyecto Java.

Para ejecutar las suites de pruebas deseadas se selecciona del contexto de ejecución (**`Run`**) la opción **`CodeCover Measurement For JUnit`**. Con ello, las medidas de cobertura serán almacenadas automáticamente en una "sesión de pruebas" establecida por CodeCover, donde se encontrarán los resultados de las pruebas ejecutadas.

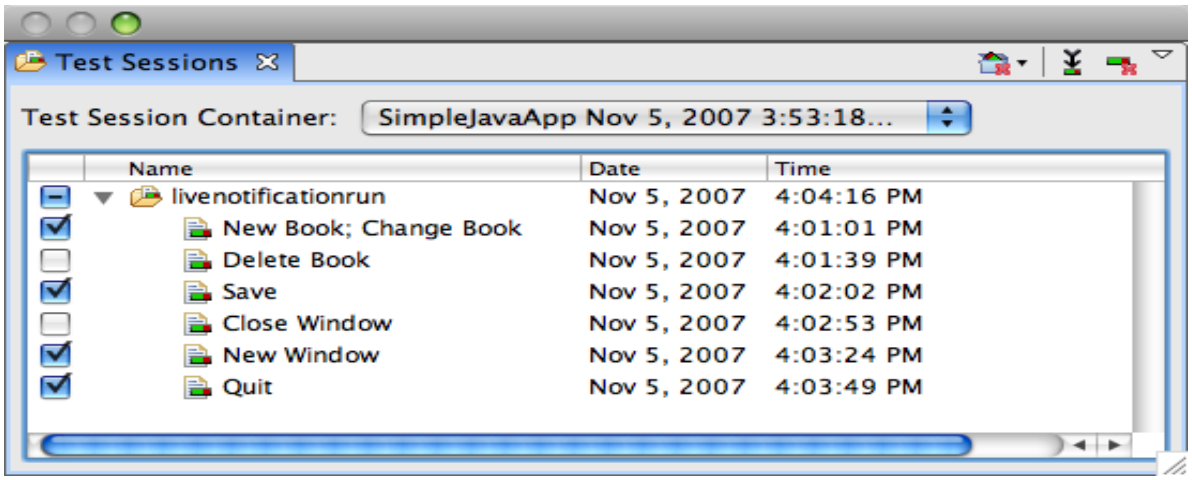

Figura 2.22 Vista de las sesiones de pruebas ejecutadas.

En la vista de las sesiones de pruebas ejecutadas se pueden seleccionar algunas de estas, además de eliminar o anexar unas con otras, según muestra la Figura 2.22. Otras vistas utilizarán las pruebas seleccionadas para mostrar otros datos.

En la Figura 2.23 se muestra la vista de cobertura donde para cada paquete, clase, método, etcétera. tendrá el porciento de cobertura del código correspondiente, a partir del tipo de cobertura que le fue analizado a cada uno, ya sean sentencias, bifurcaciones, ciclos, etcétera.

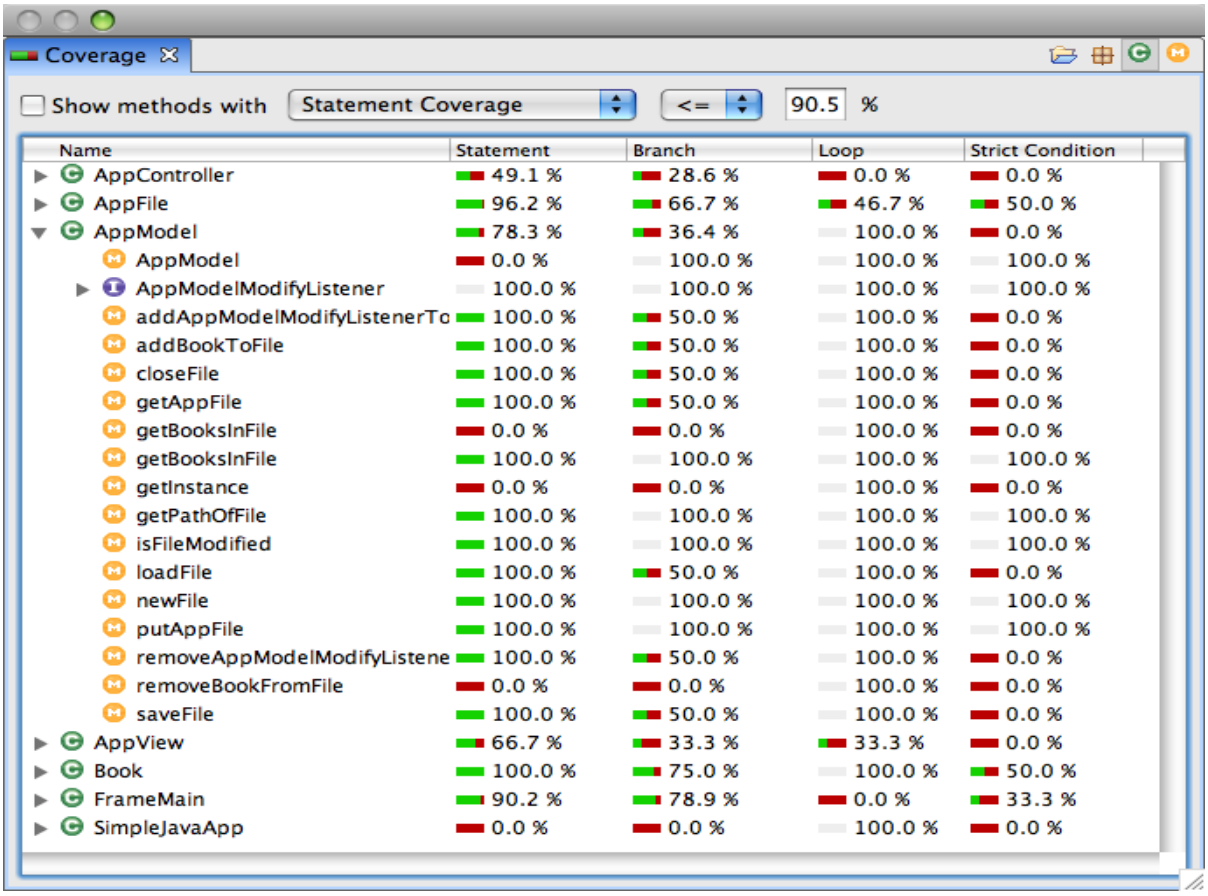

Figura 2.23 Vista de cobertura en la ejecución de las pruebas.

El código es destacado con colores acorde con los datos de las medidas obtenidas. La Figura 2.24 muestra cómo a través de los editores de código Java será visible la cobertura del código de producción, donde colores personalizados destacan las sentencias de código que han sido cubiertas o no total o parcialmente.

En la vista de la matriz de correlación, mostrada en la Figura 2.25, se comparan unas pruebas con otras. A través de cada bloque de la matriz se puede ver el grado de código que es solapado por dos pruebas a la vez. Esto ayuda a determinar cuáles pruebas pueden ser innecesarias y así eliminarlas para ahorrar tiempo de ejecución o simplemente refinarlas para que tomen otro camino en las pruebas, sin necesidad de perder cobertura del código.

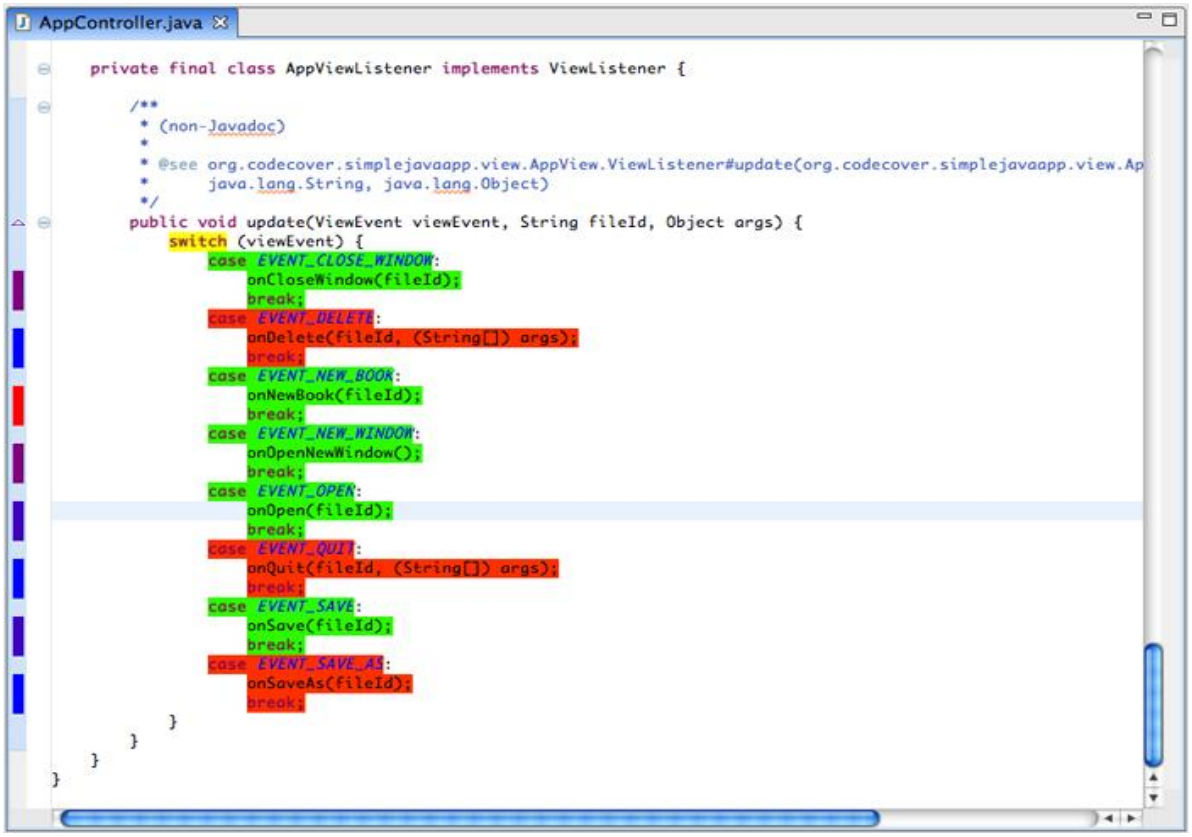

Figura 2.24 Vista de cobertura en los editores de código

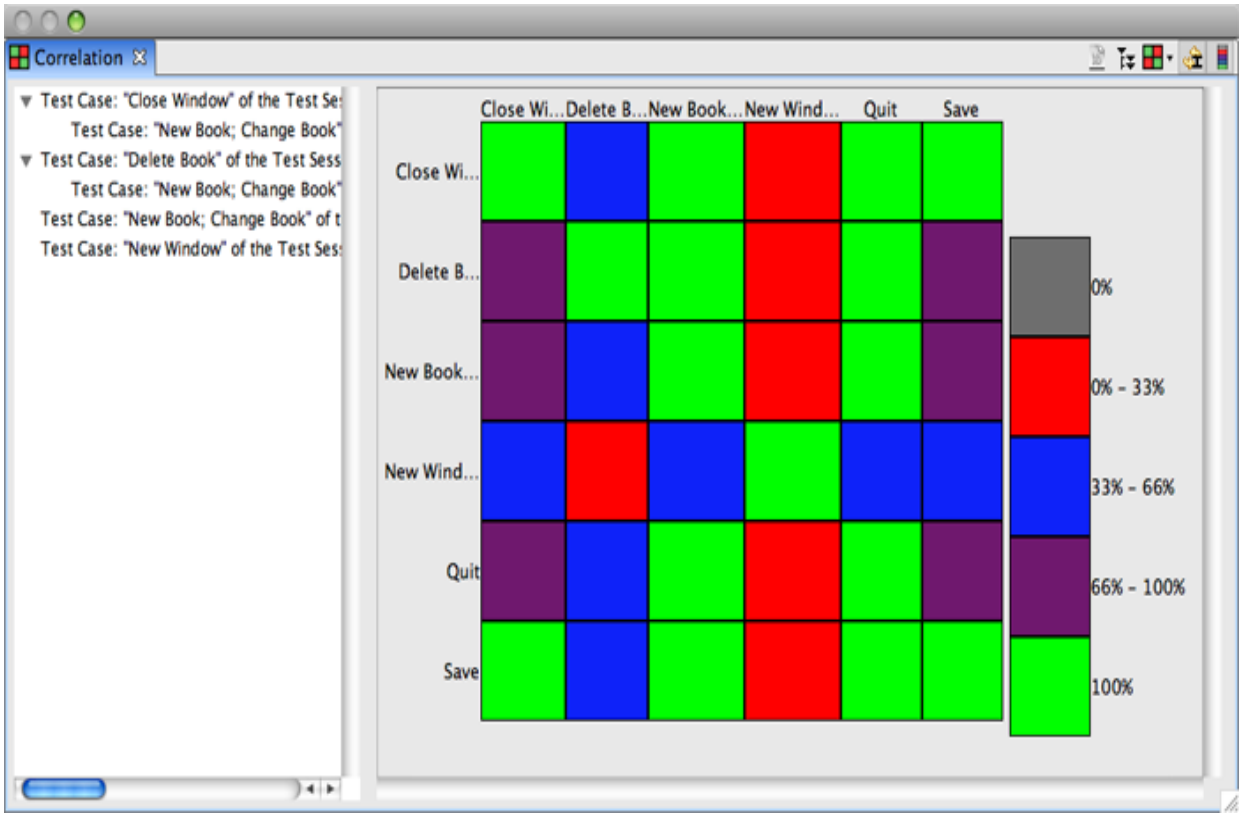

Figura 2.25 Matriz de correlación para determinar solapamiento en las pruebas.

A través de CodeCover se puede resumir y exportar los datos de las medidas de cobertura obtenidos. Los datos pueden ser obtenidos en formato CSV y HTML, además de ficheros binarios, de donde se pueden implementar generadores en Java que obtengan otros tipos de reportes a su medida, tales como se muestran en la Figura 2.26.

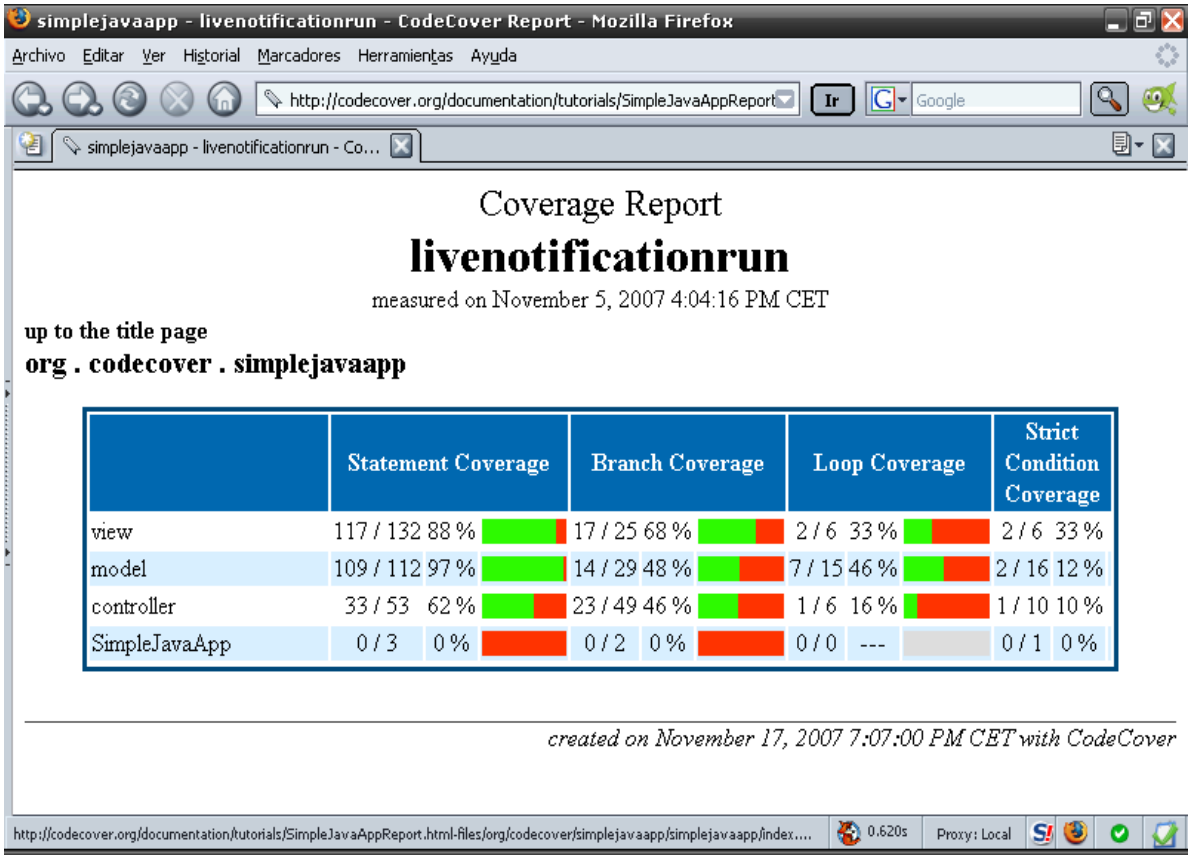

Figura 2.26 Reporte en formato HTML generado por CodeCover.

2.4 Conclusiones.

El potencial que brindan las herramientas para realizar pruebas unitarias en aplicaciones construidas en J2EE, estará dado por el adecuado uso que se les dé. Estas por sí solas no aseguran buen desempeño en la creación y ejecución de pruebas unitarias. Es imprescindible que se tenga experiencia y se adquieran habilidades que ayuden a tener buenas e ingeniosas soluciones en tal aspecto. Utilizando las técnicas adecuadas en el momento adecuado evitará al desarrollador desperdiciar recursos y aumentará su productividad.

CAPÍTULO 3: CASO DE ESTUDIO VIDEO CLUB.

3.1 Introducción.

En el capitulo anterior se mostró la filosofía del funcionamiento de las pruebas unitarias basadas en JUnit. Conjunto a ello se expusieron algunas técnicas y herramientas que son aplicadas en este tipo de pruebas. El presente capitulo procura mostrar cómo coexisten varios de estos aspectos a través de un caso de estudio.

El objetivo del mismo no es mostrar las mejores maneras de definir una arquitectura ni lograr un diseño óptimo del mismo. Más bien, se desea mostrar algunas ideas que ayuden a entender el modo en que el código de producción y sus pruebas asociadas puedan ser realizados de manera que cada uno contribuya a la mejora del otro.

3.2 Presentación del caso de estudio Video Club.

3.2.1 Descripción general del caso de estudio.

El caso de estudio que se utilizará como último paso en la instrucción que pretende ayudar a los desarrolladores, se relaciona con la gestión de los préstamos que se realizan en un Video Club a personas asociadas a él. Se controla los recursos (videocasetes) que manipula el centro y los datos relacionados a los asociados.

Las funcionalidades que se pretenden con el sistema, son las siguientes:

- Registrar y modificar los datos de un asociado.
- Registrar y modificar los datos de las películas del Video Club.
- Registrar y modificar los datos de los videocasetes con los que se cuenta, tales como las películas que en estos se encuentran.
- Registrar los préstamos y devoluciones de videocasetes realizados a los asociados.

3.2.2 Reglas de negocio del caso de estudio.

Estas reglas (RN) no son más que las descripciones de las especificaciones del sistema. Se define en ellas lo que se espera del sistema y lo que el mismo no debe permitir que se haga. Esto se convierte en aspecto principal a tomar en cuenta cuando se realizan las pruebas porque definen las

correspondientes verificaciones que deben hacerse a las funcionalidades que serán probadas. Téngase pues, la descripción de las mismas:

RN. Relacionadas al Asociado.

El registro de asociados estará dado por la introducción de los datos de los usuarios en el sistema. De cada persona se debe conocer el nombre y su número de carnet de identidad, su dirección particular como son calle, número y localidad (o reparto). El sistema generará un código aleatorio que será asignado al usuario el cual estará formado por tres letras (entre la A y la Z) y dos dígitos (entre 0 y el 9), para su futura autentificación ante el sistema. No deben existir dos usuarios con el mismo número de identidad.

Se podrán modificar solo el nombre y el número de identidad, así como los datos relacionados con su dirección particular. No se debe permitir cambiar el código asignado, ni eliminar ningún asociado del sistema.

RN. Relacionadas a las Películas.

El sistema debe permitir el registro de los datos de las películas que se encuentran archivadas y que puedan ser posteriormente agregadas en algunos videocasetes. De cada película se conocerá el título, el nombre del director o de los directores, los nombres de los actores protagónicos, el género, el año y país de producción, una sinopsis y la duración en minutos.

Se podrán editar todos los datos de una película registrada en el sistema en caso de que requieran ser modificarlos. No se podrá eliminar ninguna película del sistema.

RN. Relacionadas a los Videocasetes.

Al registrar la existencia de un videocasete en el sistema se indicará el modo en que este ha sido grabado, ya sea SP o EP que permiten hasta 120 y 360 minutos de grabación respectivamente. De las películas registradas anteriormente se seleccionarán las que han sido grabadas en el videocasete, donde el tiempo total de duración de todas debe ser menor que el tiempo permitido por el modo del videocasete correspondiente. Al editar los datos de un videocasete se puede variar el modo de grabación así como indicar las películas que hayan sido grabadas o borradas.

El sistema no debe permitir que los videocasetes sean eliminados del mismo.

RN. Relacionadas a los Préstamos.

Para efectuar un préstamo el asociado se identificará por su nombre o número de identidad, con algún funcionario del Video Club que verificará su autenticidad. El asociado seleccionará el videocasete

deseado y se registrará la fecha en que se presta, así como la cantidad de días hábiles que tendrá para efectuar la devolución. Antes de realizar un préstamo es obligatorio verificar que: el asociado tiene menos de tres préstamos en su poder y que ninguno de sus préstamos exceda la fecha límite de entrega.

Para efectuar las devoluciones se debe registrar la fecha de entrega de los videocasetes. El sistema no podrá editar ni eliminar datos relacionados con un préstamo después que este haya sido realizado.

RN. Complementarias.

No se puede prestar un videocasete que esté registrado en el sistema como prestado.

No se pueden realizar devoluciones si el videocasete no se encuentra en el sistema como prestado.

Se debe mostrar la lista de todos los asociados y realizar búsquedas de asociados en particular a partir del nombre o el número de identidad del mismo.

Se debe mostrar una lista de todas las películas registradas en el sistema y para cada una de ellas, el listado de los videocasetes que la contengan.

3.2.3 Descripción general del sistema.

El sistema será implementado en Java, utilizando la versión 6.0 de la SDK (Java Development Kit). El IDE utilizado para la implementación y ejecución de las pruebas será Eclipse en su versión 3.3 en conjunto a los plugins Spring IDE e Hibernate Tools que garantizan el trabajo con otras herramientas. La base de datos del sistema será sobre el gestor de base de datos PostgreSQL. Entre las librerías principales utilizadas están Spring 2.5, Hibernate 3.3 y Unitils 1.0, entre otras que son dependencias de las mismas.

Entidades del Sistema.

En el sistema se identifican cuatro (4) importantes entidades. Estas son **`Asociado`**, **`Pelicula`**, **`VideoCassette`** y **`Prestamo`** para representar los datos de los asociados, las películas, los videocasetes y los préstamos a registrar por el sistema. Otras entidades son **`DireccionParticular`** para contener los datos correspondientes a la dirección particular del asociado, **`ModoCassette`** para establecer los posibles modos de grabación de los videocasetes y **`GeneroPelicula`** y **`PaisProduccion`** para establecer los valores de género y país donde fue producida la película respectivamente. En la Figura 3.1 se muestra cómo están relacionadas estas clases.

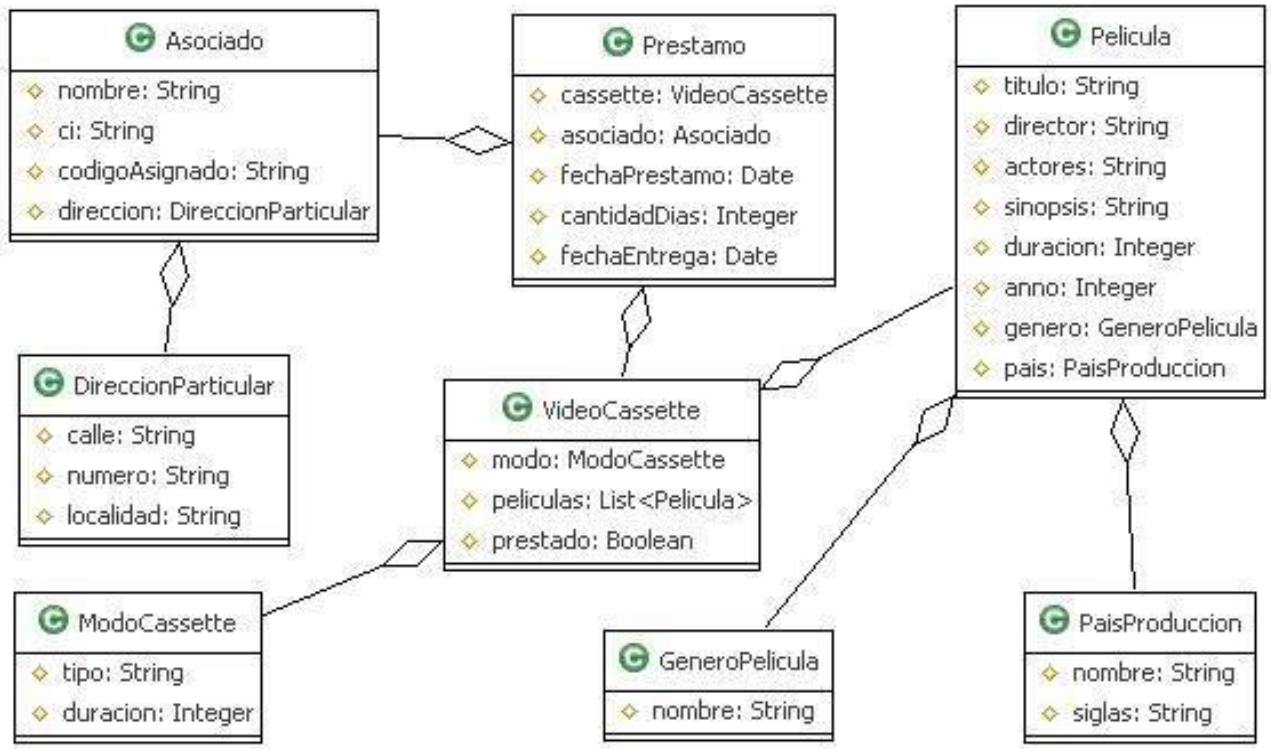

Figura 3.1 Entidades de Negocio del Sistema Video Club.

Las clases que se encargarán de mantener la lógica del negocio, la persistencia de los datos y la interacción con el usuario final, están divididas en tres capas lógicas: acceso a datos, negocio y lógica de presentación.

Diseño de la base de datos.

En la Figura 3.2 se muestra el diseño de la base de datos creada para mantener el estado del sistema. Las tablas **`ASOCIADO`**, **`PELICULA`**, **`CASSETTE`** y **`PRESTAMO`** representan los datos de las entidades del sistema. Otras tablas como **`MODO_CASSETTE`**, **`DIRECCION`**, **`PAIS_PRODUCCION`** y **`GENERO_PELICULA`** almacenarán datos relacionados con las tablas anteriores. La tabla **`CASSETTE_PELICULA`** establece la relación que existirá entre un videocasete y las películas grabadas en él.

Controladores, Managers, Validadores y DAOs.

Subclases de la interface **`Controller`**, del framework Spring MVC, se utilizan para implementar la lógica de presentación del flujo web. Una interface servirá de fachada para separar la lógica de presentación con el resto de la lógica del negocio implementada en el sistema, para convertir a este en una especie de caja negra. Y dentro de esta capa se tendrán otras dos, una de **`Managers`** y otra de

DAOs, clases que implementarán las acciones correspondiente al negocio y al acceso a datos por entidades o funcionalidades, respectivamente.

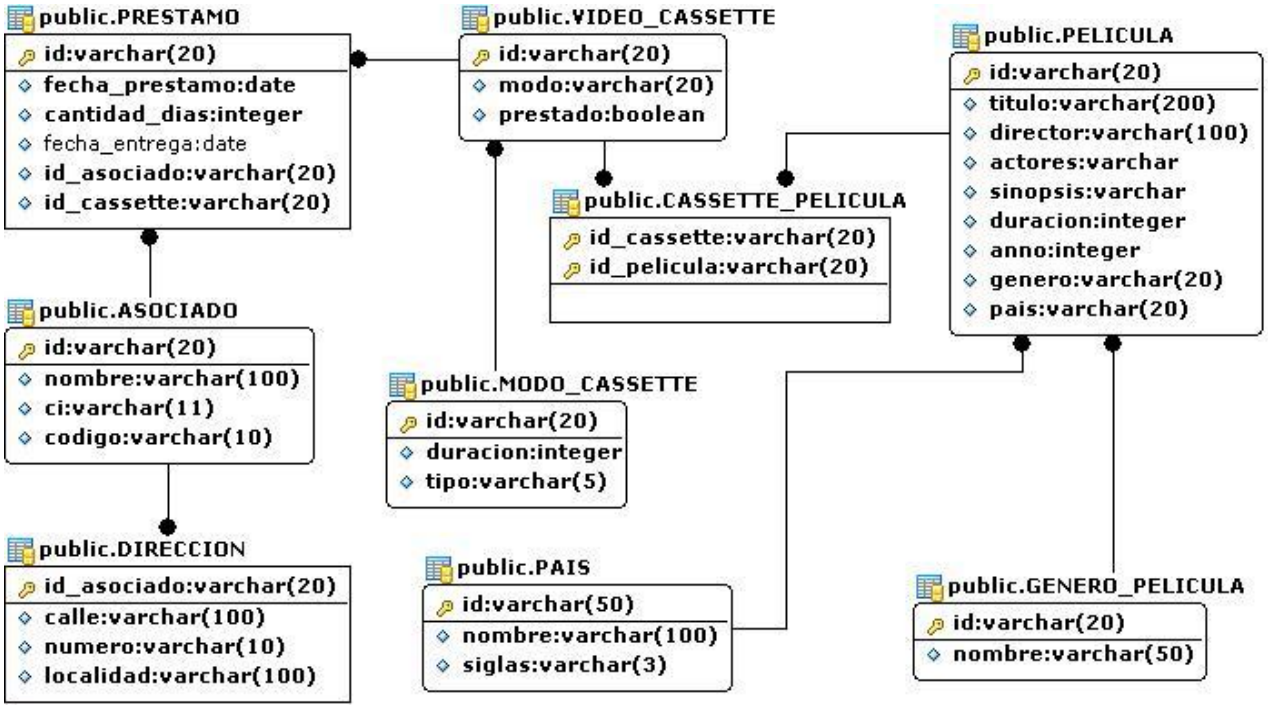

Figura 3.2 Diseño de la Base de Datos para el sistema Video Club.

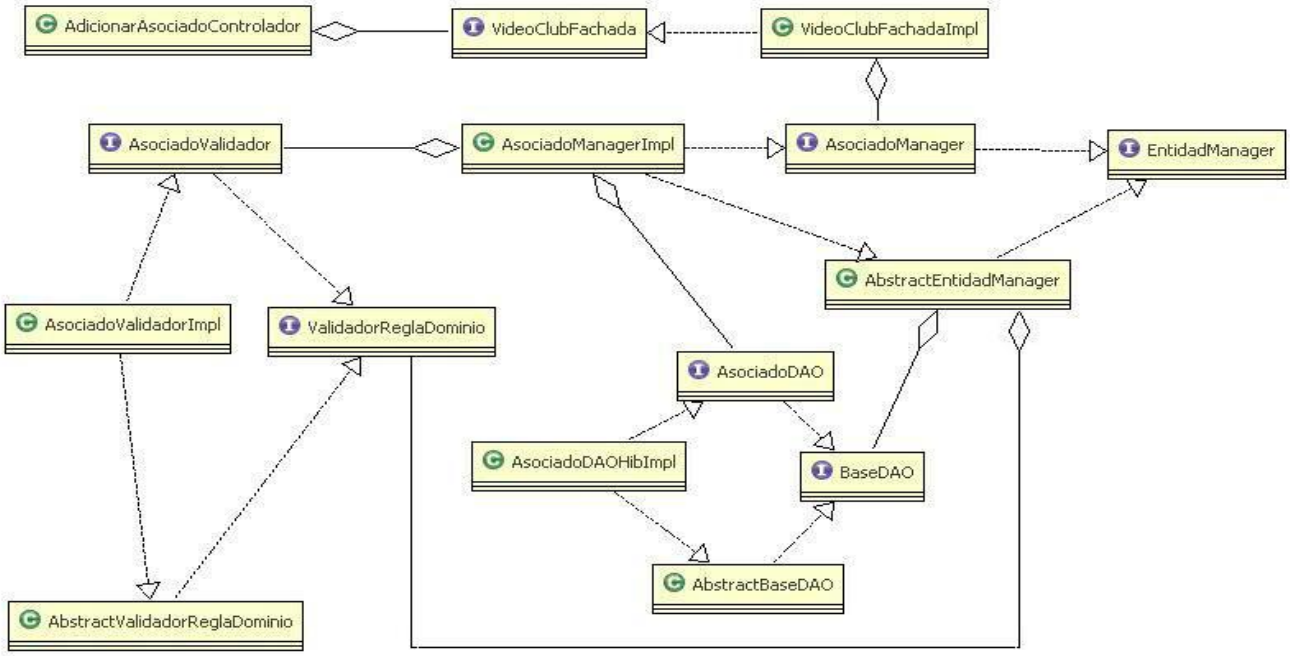

Figura 3.3 Relación entre las clases de las capas de Presentación, Lógica de Negocio y Acceso a Datos.

Los **`Managers`** también se apoyan en **`Validadores`** que verifican las condiciones que deben cumplirse antes, durante o después de realizar acciones que cambien el estado del sistema. En la Figura 3.3, se muestran las relaciones de jerarquía y colaboraciones existentes entre los controladores, los managers, los daos y los validadores para el caso de las funcionalidades del sistema relacionadas con los asociados.

3.3 Herramientas utilizadas en el caso de estudio.

JUnit ha evolucionado desde su creación para lograr potencialidades y flexibilidad en el framework. El uso de las anotaciones para dar soporte en la introducción de aspectos en el código de prueba, es un ejemplo de estos avances.

Otro hecho ha sido el surgimiento de frameworks que han revolucionado el desarrollo de aplicaciones construidas en J2EE tales como Spring e Hibernate. Estos también han motivado el surgimiento y desarrollo de otros que acaparan varias técnicas y herramientas para ofrecer soluciones integrales en las pruebas unitarias tales como Unitils.

3.3.1 Framework de aplicación Spring para soluciones J2EE.

Spring es un framework de codigo abierto que provee una manera de gestionar los objetos del sistema. Se estructura por capas para permitir la selección y el mantenimiento de cualquier parte de manera aislada, siempre que su estructura sea internamente consistente (WALLS and BREIDENBACH 2005). Esto, por ejemplo, permite emplear a Spring para simplificar el uso de código JDBC o sólo para manejar los objetos de negocio. Está diseñado desde su concepción para ayudar a escribir código que sea fácil de probar, siendo un framework ideal para proyectos dirigidos por las pruebas.

Rob Johnson, acerca de Spring, menciona las siguientes ideas: “Spring es un framework de aplicación de código abierto cuyo objetivo es hacer el desarrollo en J2EE más sencillo y fácil (…). El objetivo de Spring es ayudar a estructurar aplicaciones enteras de una manera productiva y consistente, agrupando lo mejor de un conjunto de frameworks y crear una arquitectura coherente. (…) Spring es a la vez el más popular y el más ambicioso de los frameworks de peso ligero. (…) Es el único que puede encargarse de todas las capas de una típica aplicación en J2EE y el único que ofrece un rango de servicios, como contenedor de peso ligero que es.” (JOHNSON, ROD *et al.* 2005).

Spring cuenta con una serie de módulos que proveen potenciales funcionalidades para una aplicación J2EE. Entre los más notables se encuentran los siguientes:

- **Contenedor de Inversión de Control** – este es el contenedor principal que provee Spring. Posibilita una configuración sofisticada del manejo de POJOs (*Plain Old Java Objects*) y

gestiona el acople entre ellos, estén sencilla o fuertemente acoplados. Trabaja con otras partes de Spring para brindar servicios como gestor de configuración que es.

- **Framework de programación orientada a aspectos** (AOP) – la programación orientada a aspectos brinda comportamientos a través de métodos que pueden ser modularizados en un solo lugar. Spring usa AOP para proveer servicios importantes tales como el manejo de transacciones o personalizar código que de otra manera estaría disperso entre clases de la aplicación.
- **Abstracción en el acceso a datos** – Spring propone una ventaja arquitectónica consistente para el acceso a datos y una singular y poderosa abstracción para implementarla. Provee de una rica jerarquía de excepciones de acceso a datos independientes de cualquier producto de persistencia. Cuenta con un rango de servicios que ayudan a lidiar con APIs de persistencia, posibilitando que los desarrolladores escriban codigo persistente a través de interfaces definidas y luego implementando con la herramienta seleccionada.
- **Framework MVC**– provee un framework basado en el patrón MVC[37] para el manejo de peticiones web. Es usado para compartir instancias de controladores multihilos de manera flexible, similar al contenedor de inversión de control. Por sus características Spring puede incluso ser usado con otros frameworks web tales como Struts o JSF (Java Server Faces).

De manera general se puede decir que Spring es un framework no invasivo. Provee un modelo de programación consistente a cualquier entorno. Tiene como objetivo facilitar el diseño orientado a objeto en aplicaciones J2EE. Promueve la buena práctica de programar a interfaces más que a clases. Facilita extraer configuraciones del código Java y declararlas en ficheros XML. Es diseñado para que las aplicaciones que lo usen sean tan fáciles de probar como sea posible.

3.3.2 Framework Hibernate para la persistencia de datos.

Hibernate es una herramienta de mapeo objeto-relacional [38] para la plataforma Java. Facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos XML. Hibernate busca solucionar el problema de la diferencia entre

---

[37] El patrón Modelo-Vista-Controlador (MVC) se refiere a separar un objeto –el modelo- de los elementos de interface de usuario que lo describen –la vista y el controlador. (…) Los creadores del patrón MVC visionaron que la apariencia de un componente (su vista) puede ser separada de su percepción (su controlador).(METSKER, S. J. *Design Patterns Java™ Workbook*, Addison Wesley, 2002..

[38] El término mapeo objeto-relacional (ORM siglas en inglés de object/relational mapping) se refiere a la técnica de mapear la representación de datos de un modelo de objetos a un modelo de datos relacional con esquemas basados en SQL.

estos dos modelos: el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional).(BAUER and KING 2007)

Permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información, Hibernate le permite a la aplicación manipular los datos de la base de datos operando sobre objetos con todas las características de la POO. Hibernate convierte los datos entre los tipos utilizados por Java y los definidos por SQL. Genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todas las bases de datos con un ligero incremento en el tiempo de ejecución. Ofrece también un lenguaje de consulta de datos llamado HQL (Hibernate Query Language) y una API, conocida como Criteria, para construir las consultas.

“La meta de Hibernate es aliviar al desarrollador de un 95 por ciento de las tareas programadas relacionadas con persistencia de datos. Hibernate no puede ser la mejor solución para aplicaciones centrada en los datos que sólo usan procedimientos almacenados para implementar lógica de negocio en la base de datos, es más útil con modelos de dominio orientado a objetos y lógica de negocio en la capa media del código en Java. Sin embargo, Hibernate puede ayudar a eliminar o encapsular código SQL y ayudará con las tareas comunes de traducción de conjuntos de resultados en representación tabular a un grafo de objetos.”(RED HAT 2008)

3.3.3 Framework Unitils para soluciones integrales de pruebas.

Unitils es una librería de código abierto con el propósito de hacer las pruebas unitarias fáciles y posibles de mantener. Unitils se basa en librerías existentes tales como DbUnit y EasyMock, integrándose con JUnit y TestNG. Provee utilidades de aserciones generales, soporta pruebas para bases de datos y el uso de objetos falsos ofreciendo integración con Spring, Hibernate y el API de Persistencia de Java (JPA). Está diseñado para ayudar en las pruebas unitarias de forma configurable y poco acoplada.(VERMEIREN *et al.* 2008).

Entre las características que presenta están las siguientes:

- **Utilidades generales para las pruebas** – aserciones de igualdad a través de técnicas de reflexión de Java, con diferentes opciones tales como ignorar los valores nulos o el orden en las colecciones.
- **Utilidades para pruebas a bases de datos** – mantenimiento automático y desactivación de restricciones para las pruebas de bases de datos (soporte para Oracle, Hsqldb, MySql, DB2, PostgreSQL y Derby). Simplifica la configuración de conexión para las pruebas. Hace simple la inserción de datos de pruebas con DbUnit y ejecuta las pruebas en un contexto transaccional.

Se integra con Hibernate, gestiona la sesión y verifica automáticamente el mapeo de los objetos con la base de datos.

- **Utilidades de objetos falsos** – simplifica la creación de objetos falsos con EasyMock y permite la inyección de dependencias usando técnicas de reflexión en Java.
- **Integración con Spring** – permite la configuración del **`ApplicationContext`** de Spring y una sencilla inyección de los objetos que gestiona en la prueba unitaria. Soporta el uso de la **`SessionFactory`** de Hibernate en configuraciones de Spring para la ejecución de las pruebas.

3.4 Solución dada en las pruebas unitarias.

En los próximos epígrafes se mostrarán algunas consideraciones para realizar las pruebas en aplicaciones construidas en J2EE usando Spring, Hibernate y Unitils. El propósito será mostrar algunas funcionalidades que brindan estas herramientas y que bien pueden ser aprovechadas en la realización y ejecución de las pruebas unitarias.

3.4.1 Pruebas basadas en Unitils.

Habilitar las pruebas con Unitils.

Una primera variante, para habilitar Unitils en las pruebas unitarias, es que la clase de prueba sea subclase de **`UnitilsJUnit3`** o de **`UnitilsJUnit4`**, dependiendo de la versión a utilizar: **`JUnit3.x`** o **`JUnit4.x`**, respectivamente. Estas clases son subclases de **`TestCase`** y contienen la lógica que le permite a Unitils adentrarse en la ejecución de las pruebas.

La segunda de estas variantes es anotar la clase con **`@RunWith`** e indicar el **`TestRunner`** como parámetro necesario para ejecutar las pruebas, ya sea **`UnitilsJUnit3TestClassRunner`** o **`UnitilsJUnit4TestClassRunner`**. (Anexos I, II y III).

Utilidades para las aserciones.

Una típica prueba contiene una parte en la que se comparan los valores de los resultados de la prueba con valores esperados. Unitils ofrece utilidades de aserciones para ayudar a reducir el código correspondiente a esta actividad. Por ejemplo, cuando se comparan dos objetos con el métodos tales como **`assertEquals()`**, este responde según la implementación del método **`equals()`**, correspondiente a la clase a la que pertenecen dichos objetos. El problema radica en que obliga a

redefinir este método en la clase y peor aún, si las estrategias de comparación cambian entre una prueba y otra, no se puede evitar que la comparación en uno de los dos casos falle.

Unitils ofrece mejoras en la forma de chequear igualdades entre los datos obtenidos y los esperados a través de técnicas de reflexión en Java. Usando métodos de la clase **`ReflectionAssert`** tales como **`assertRefEquals()`**, **`assertLenEquals()`**, **`assertPorpertyLenEquals()`** y **`assertPropertyRefEquals()`**. A dichos métodos no sólo puede indicársele lo que verificarán, sino también cómo lo harán: por ejemplo, que no tomen en cuenta el orden en un conjunto de valores dados o los valores nulos entre las propiedades de objetos, entre otros. (Anexos I, II y III).

3.4.2 Usando el contexto de Spring.

Configuración del ApplicationContext de Spring.

Cargar un contexto de la aplicación puede hacerse de manera sencilla anotando la clase de prueba con **`@SpringApplicationContext`**, especificando como parámetros los nombres de los ficheros de configuración.

Otra manera de indicar el contexto es creando un método que no reciba parámetros y que retorne una instancia de **`ConfigurableApplicationContext`** donde se construya el contexto, definiendo explícitamente cuáles son los ficheros correspondientes. Este método debe anotarse con **`@SpringApplicationContext`** para que sea reconocido como tal.

Los atributos de la clase de prueba, que deban ser referencias a objetos creados en el contexto de la aplicación, pueden ser anotados con **`@SpringBean`**, **`@SpringBeanByType`** o **`@SpringBeanByName`**. Según el caso seleccionado, a la prueba le será inyectado el objeto del contexto, ya sea por el nombre o el tipo. Las mismas anotaciones pueden ser utilizadas en métodos de tipo **`setXXX()`** de los atributos correspondientes. (Anexos I y III).

3.4.3 Pruebas a código persistente basado en Hibernate.

Spring brinda funcionalidades similares para la carga del contexto y la inyección de los objetos necesarios para ejecutar las pruebas. La ventaja de Unitils sobre esto radica en que está integrado con otras herramientas de prueba como DbUnit, permitiendo hacer lo mismo conjunto a otros frameworks. Las pruebas a la base de datos deben ser ejecutadas en una base de datos de prueba, donde se pueda tener completo control de los datos de prueba a ser usados. (Anexo I y III).

Cargando el DataSet a través de ficheros de datos.

Para que al ser ejecutada la prueba se carguen los datos necesarios, se anota con **`@DataSet`** un método de prueba (carga los datos para dicho método) o la clase de prueba (carga los datos para cada método de prueba). Si es anotada la clase, Unitils tratará de encontrar algún fichero de datos localizado en el mismo paquete que la clase de prueba y cuyo nombre sea el mismo que la clase. De manera similar, en caso de ser anotado un método el fichero de datos debe tener como nombre el de la clase más el del método, separados por un punto (.). Además, se puede especificar uno o más nombres de ficheros distintos si es que así se desea, como propiedad de la anotación **`@DataSet`**. Esto hace que la base de datos sea rellenada con los datos del fichero antes de ejecutar la prueba (Anexos I y III).

Probar el mapeo de Hibernate.

Unitils contiene una prueba para los ficheros de mapeos de Hibernate, simple pero poderosa. Esta prueba verifica que los mapeos son consistentes con la base de datos. Para activarla, solo es necesario llamar al método estático **`assertMappingWithDatabaseConsistent()`** de la clase **`HibernateUnititls`**. De encontrar algún error, este hará fallar la prueba mostrando un mensaje que diga cuáles tablas y/o qué columnas deben ser creadas o modificadas. (Anexos I y III).

Verificar datos desde un DataSet.

Las pruebas de elementos con código persistente en ocasiones necesitan verificar que el estado de los datos en la base de datos coincide con ciertos datos esperados. Estos datos esperados pueden encontrarse configurados en un fichero de datos. Para ello puede utilizarse el método **`DbUnitAssert.assertEqualsDataSet()`** que verificará la igualdad entre dos **`DataSets`** o el método **`DbUnitAssert.assertDbContentAsExpected()`** para verificar la igualdad entre un ***DataSet*** y los datos en la base de datos. (Anexos I y III)

3.4.4 Pruebas con objetos falsos creados con EasyMock.

Anteriormente, se mencionaron dos técnicas usadas para aislar los componentes y que puedan ser probados sin necesidad de depender de otros componentes que no estén disponibles. En el caso de los objetos falsos, Unitils brinda utilidades para el trabajo con EasyMock, simplificando el proceso de creación, de inyección y de configuración de estos. (Anexo II)

Creación e inyección de Objetos Falsos.

Cuando la prueba utilice objetos falsos, estos pueden ser atributos suyos, así como lo puede ser un objeto de la clase que será probada. En tales casos, el atributo objetivo de la prueba se anota con

@TestedObjetc indicando así cual será la entidad bajo prueba. Los restantes atributos, de los cuales la clase bajo prueba dependerá se anotan con **@Mock**, indicando cuales serán convertidos en objetos falsos. Con las anotaciones **@InjectInto** y **@InjectIntoByType** a los objetos falsos, se puede especificar a qué propiedad del objeto a ser probado deben ser inyectados. Estos objetos falsos serán automáticamente creados por Unitils y serán instalados en el objeto a ser probado antes de comenzar el método de prueba. (Anexo II)

Declarando expectativas de los objetos falsos.

Los objetos falsos atenderán peticiones del objeto a ser probado a través de llamadas a algunos de sus métodos definidos en la clase o interface determinada. EasyMock permite un mecanismo para definir los comportamientos de estos objetos falsos.

El proceso consiste en recrear la llamada de cada uno de estos métodos e indicar los valores que deben retornar si es el caso. Cuando el método de un objeto falso no devuelve valores (es de tipo **void**) solo se llamará al método y se especificarán cuales son los parámetros que debe esperar recibir. Si el método a describir retorna valores, se definirá dentro de **EasyMock.expected()** con los correspondientes parámetros esperados y a esto se agregará **.andReturn()** para indicar el valor a ser retornado, quedando de la forma **EasyMock.expected().andReturn()**.

De esta manera queda grabado el comportamiento esperado por todos los objetos. Al hacer la llamada **EasyMockUnitils.replay(),** se activa el **replay()** de todos los objetos falsos preparándolos para responder con el comportamiento indicado. Terminada la ejecución de la prueba, es invocado el método **EasyMockUnitils.verify()** automáticamente, verificando en cada objeto falso si se ha cumplido el comportamiento esperado. (Anexo II)

3.4.5 Pruebas de integración de componentes.

Las pruebas de integración en ocasiones pueden considerarse también pruebas unitarias. La unión de varios componentes para entre todos responder a una funcionalidad del sistema puede considerarse como una unidad (menos atómica) respecto a todo el sistema. En ocasiones estos componentes representan varias capas del sistema, tales como acceso a datos y lógica del negocio. No tiene sentido utilizar agentes dobles sino las implementaciones reales para verificar que lo que será probado funcionará como se requiere.

En este tipo de pruebas es recomendable usar en conjunto las herramientas mencionadas anteriormente. Las utilidades que proporciona Unitils (integradas con otros frameworks como DbUnit), unidas a Spring para cargar los componentes definidos en su contexto y con Hibernate para la

persistencia de los datos, hará posible una prueba de integración que no necesite muchos recursos. (Anexo III)

3.5 Conclusiones.

El uso de frameworks y librerías como Spring, Hibernate y Unitils que brindan funcionalidades para potencializar la productividad durante el desarrollo, es de vital importancia. Estos abstraen y reducen el código de implementaciones engorrosas que pueden introducir consecuentemente errores en sistema. Unido a ello, es conveniente tener en cuenta las técnicas y buenas prácticas que proponen hacer un código que sea fácil de probar. Esto ayudará en buena medida a que el equipo de desarrollo alcance la calidad en el producto que se espera.

CONCLUSIONES.

En el trabajo presentado se instruye sobre la importancia que implica para los desarrolladores hacer pruebas unitarias al código de producción. Conocer las variantes en que estas pueden ser escritas y lo que se puede lograr en cada momento, para ayudar a tomar decisiones estratégicas de cómo y cuándo realizarlas. Además se mostraron herramientas que son posibles usar durante la creación y ejecución de pruebas unitarias en la plataforma J2EE. A través de ejemplos concretos se expusieron algunas técnicas y buenas prácticas comúnmente utilizadas en este tipo de actividad durante el desarrollo del producto.

Las pruebas unitarias siempre contribuirán al trabajo de los desarrolladores. En ellos está el deber entonces, de hacerlas parte de su cotidianidad y de las posibilidades con las que contar para enfrentar el desarrollo de manera exitosa. Siempre será más útil emplear parte del tiempo en hacer y ejecutar una prueba que en gastarlo descubriendo errores de otra forma. Es importante conocer cuáles puedan ser las herramientas adecuadas, a partir de la tecnología con la que se cuenta y se desarrolla el sistema. No basta sólo con dominar una u otra herramienta, sino también saber emplearlas en el momento y el modo correctos, potenciando así un uso efectivo.

RECOMENDACIONES.

Se recomienda que los desarrolladores apliquen los aspectos tratados en el trabajo de modo que puedan hacer uso de las pruebas unitarias como parte de su labor en el proceso de creación del producto.

Estudiar las actividades de otras áreas del proceso de desarrollo de software que puedan contribuir en la creación y ejecución de las pruebas unitarias, tales como la confección de los casos de prueba a partir de los requerimientos del sistema o la gestión de los riesgos para determinar la factibilidad de hacer las pruebas unitarias en un momento dado.

Desarrollar un trabajo similar teniendo en cuenta otras tecnologías y herramientas para que proyectos que no desarrollan en la plataforma J2EE cuenten con un material que guíe a sus desarrolladores a crear y ejecutar pruebas unitarias.

REFERENCIA BIBLIOGRÁFICA.

BAUER, C. and G. KING. *Java Persistence with Hibernate*, Manning Publications Co., 2007.

IEEE, I. *IEEE Standard for Software Test Documentation*. SOCIETY, S. E. T. C. O. T. I. C., 1998. IEEE Std 829-1998.

JOHNSON, R. *Expert One-on-One J2EE Design and Development*, Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

JOHNSON, R. and J. HOELLER. *Expert One-on-One™ J2EE™ Development without EJB™*, Wiley Publishing, Inc., Indianapolis, Indiana, 2004.

JOHNSON, R.; J. HOELLER, *et al. Professional Java Development with the Spring Framework*, Wiley Publishing, Inc., 2005.

LEWIS, W. E. *Software Testing and Continuous Quality Improvement TEAM*, AUERBACH PUBLICATIONS, 2005.

MESZAROS, G. *xUnit Test Patterns - Refactoring Test Code*, Addison Wesley, 2007.

METSKER, S. J. *Design Patterns Java™ Workbook*, Addison Wesley, 2002.

RED HAT, I. *Hibernate Reference Documentation*, 2008. Version: 3.2.6.

VERMEIREN, B.; F. BEERNAERT, *et al. Unitils*, 2008. [Disponible en: http://www.unitils.org/summary.html

WALLS, C. and R. BREIDENBACH. *Spring in Action*, Manning Publications Co., 2005.

BIBLIOGRAFÍA

BACH, J. Heuristic Risk-Based Testing. Software Testing and Quality Engineering Magazine, 1999. 11/99: 9.

BAUER, C. and G. KING. Hibernate in Action, Manning Publications Co., 2005.

BAUER, C . Java Persistence with Hibernate, Manning Publications Co., 2007.

BECK, K. Test-Driven Development By Example, Addison Wesley, 2002.

BEIZER, B. Black-Box Testing - Techniques for functional testing of software and systems, John Wiley and Sons, 1995.

BEIZER, B. Software Testing Techniques, Van Nostrand Reinhold, 1990.

BEUST, C. and H. SULEIMAN. Next Generation Java™ Testing, Addison Wesley, 2008.

BINDER, R. V. Testing Object-Oriented Systems : Models, patterns and tools, Addison-Wesley, 2000.

COHEN, F. Java Testing and Design, Prentice Hall, 2004.

DOAR, M. B. Practical Development Environments, O'Reilly, 2005.

DUSTIN, E.; J. RASHKA, et al. Automated Software Testing : Introduction, management and performance, Addison-Wesley, 2000.

FOWLER, M.; K. BECK, et al. Refactoring: Improving the Design of Existing Code, Book News, Inc., 2002.

FRANKE, S.; R. HANUSSEK, et al. CodeCover, 2008. [Disponible en: http://codecover.org/

GIACCO, R. L.; F. LEME, et al. DbUnit Project, 2008. [Disponible en: http://www.dbunit.org/

GOLD, R. HttpUnit - ServletUnit, 2008. [Disponible en: http://httpunit.sourceforge.net/index.html

GRAHAM, D. and M. FEWSTER. Software Test Automation, Addison-Wesley, 1999.

HAMMELL, T.; R. GOLD, et al. Test Driven Development: A J2EE Example. APRESS, 2005.

HOFFMANN, M. R. EclEmma - Java Code Coverage for Eclipse, 2008. [Disponible en: http://www.eclemma.org/

IEEE, I. IEEE Standard for Software Test Documentation. SOCIETY, S. E. T. C. O. T. I. C., 1998. IEEE Std 829-1998.

JANZEN, D. S. Software Architecture Improvement thru Test-Driven Development: An Empirical Study, University of Kansas, 2005.

JENDROCK, E.; J. BALL, et al. The Java™ EE 5 Tutorial Third Edition, Sun Microsystems, Inc, 2006.

JOHNSON, R. Expert One-on-One J2EE Design and Development, Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

JOHNSON, R. and J. HOELLER. Expert One-on-One™ J2EE™ Development without EJB™, Wiley Publishing, Inc., Indianapolis, Indiana, 2004.

JOHNSON, R.; J. HOELLER, et al. Professional Java Development with the Spring Framework, Wiley Publishing, Inc., 2005.

KANER, C. Paradigms of Black Box Software Testing, Florida Institute of Technology, 2002.

KIT, E. Software Testing in the Real World - improving the process, Addison-Wesley 1995.

LADD, S.; D. DAVISON, et al. Expert Spring MVC and Web Flow. APRESS, 2006.

LEWIS, W. E. Software Testing and Continuous Quality Improvement TEAM, AUERBACH PUBLICATIONS, 2005.

LINK, J. and P. FROHLICH. Unit Testing in Java: How Tests Drive the Code, MORGAN KAUFMANN 2002.

MARSCHALL, P. Detecting the Methods under Test in Java, Software Composition Group University of Bern, 2005.

MARTIN, R. C. RUP®/XP Guidelines: Test-first Design and Refactoring. Rational Software White Paper, Rational, 2001.

MASSOL, V. and T. HUSTED. JUnit in Action, Manning Publications, Co., 2004.

MESZAROS, G. xUnit Test Patterns - Refactoring Test Code, Addison Wesley, 2007.

METSKER, S. J. Design Patterns Java™ Workbook, Addison Wesley, 2002.

PEAK, P. and N. HEUDECKER. Hibernate Quickly, Manning Publications Co., 2006.

PERRY, W. E. Effective Methods for Software Testing, John Wiley and Sons, 2000.

PETTICHORD, B.; C. KANER, et al. Lessons Learned in Software Testing, John Wiley and Sons 2002.

PEZZE, M. and M. YOUNG. Software Testing and Analysis, John Wiley & Sons, Inc., 2008.

RED HAT, I. Hibernate Reference Documentation, 2008. Version: 3.2.6.

SIEGEL, S. Object Oriented Software Testing: A Hierarchical Approach, John Wiley and Sons, 1996.

VEENENDAAL, E. V.; M. POL, et al. Software Testing - A Guide to the Tmap Approach, Addison-Wesley, 2002.

VERMEIREN, B.; F. BEERNAERT, et al. Unitils, 2008. [Disponible en: http://www.unitils.org/summary.html

WALLS, C. and R. BREIDENBACH. Spring in Action, Manning Publications Co., 2005.

ANEXO I – PRUEBA A CÓDIGO PERSISTENTE BASADO EN HIBERNATE UTILIZANDO UNITILS Y DBUNIT.

Prueba a código persistente basado en Hibernate.

Clase AsociadoTest probando código persistente basado en Hibernate

```
package cu.uci.tesis.capitulo3.videoclub.dao;
import java.util.ArrayList;
import java.util.List;
import javax.sql.DataSource;
import junit.framework.Assert;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.junit.Test;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.unitils.UnitilsJUnit4;
import org.unitils.dbunit.annotation.DataSet;
import org.unitils.dbunit.util.DbUnitAssert;
import org.unitils.dbunit.util.DbUnitDatabaseConnection;
import org.unitils.orm.hibernate.HibernateUnitils;
import org.unitils.reflectionassert.ReflectionAssert;
import org.unitils.reflectionassert.ReflectionComparatorMode;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import cu.uci.tesis.capitulo3.nucleo.dao.SearchCriteria;
import cu.uci.tesis.capitulo3.nucleo.dao.SearchCriteria.ComparationType;
import cu.uci.tesis.capitulo3.videoclub.dominio.Asociado;

@SpringApplicationContext( {
"classpath:cu/uci/tesis/capitulo3/videoclub/configuracion/cu-uci-tesis-capitulo3-videoclub-context-test.xml",
"classpath:cu/uci/tesis/capitulo3/videoclub/configuracion/cu-uci-tesis-capitulo3-videoclub-dataaccess-context.xml" })
@DataSet("AsociadoDataSet.xml")
public class AsociadoDAOTest extends UnitilsJUnit4 {
```

```
	@SpringBeanByType
	private AsociadoDAO asociadoDAO;
	@SpringBeanByType
	private DataSource dataSource;
	@BeforeTest
	public void setUp() {
	}
	@AfterTest
	public void tearDown() {
	}
	@DataSet("AsociadoDataSet.xml")
	@Test
	public void testFindAll() throws Exception {
		List<Asociado> esperados = new ArrayList<Asociado>();
		esperados.add(new Asociado("10001", "Ana Laura",
							"88041301234", "ABC05"));
		esperados.add(new Asociado("10003", "Pedro Pablo",
							"71060690007", "MNO42"));
		esperados.add(new Asociado("10002", "Miguel Angel",
							 "82110956754","XYZ21"));
		List<Asociado> asociados = asociadoDAO.findAll();
		ReflectionAssert.assertRefEquals(esperados, asociados,
				ReflectionComparatorMode.IGNORE_DEFAULTS,
				ReflectionComparatorMode.LENIENT_ORDER);
	}
	@Test
	public void testFindAllBySearchCriterias() throws Exception {
		List<Asociado> esperados = new ArrayList<Asociado>();
		esperados.add(new Asociado("10001", "Ana Laura",
							"88041301234", "ABC05"));
		esperados.add(new Asociado("10002", "Miguel Angel",
							"82110956754", "XYZ21"));
		SearchCriteria criterioCI = new SearchCriteria("ci",
				ComparationType.CONTAINS, "8");
		List<SearchCriteria> criterios = new ArrayList
							<SearchCriteria> ();
```

```
        criterios.add(criterioCI);
        List<Asociado> asociados = asociadoDAO
                    .findAllBySearchCriterias(criterios);
        ReflectionAssert.assertRefEquals("deben ser 3 asociados",
         esperados, asociados, ReflectionComparatorMode.IGNORE_DEFAULTS,
                                  ReflectionComparatorMode.LENIENT_ORDER);
    }
    @Test
    public void testFindByIdNull() throws Exception {
        Asociado asociado = asociadoDAO.findById("AS122", false);
        Assert.assertNull("No debe existir ID [AS122]", asociado);
    }
    @Test
    public void testFindByIdOK() throws Exception {
        Asociado asociado = asociadoDAO.findById("10002", false);
        Assert.assertNotNull("Debe existir asociado con ID [10002]",
                                                                asociado);
    }
    @Test
    public void testHibernateMapping() {
        HibernateUnitils.assertMappingWithDatabaseConsistent();
    }
    @Test
    @DataSet("AsociadoPersistDataSet.xml")
    public void testInsert() throws Exception {
        Asociado asociado = new Asociado("Manolo Martinez",
                                                "65121295022","TRS34");
        asociadoDAO.persist(asociado);
        IDataSet expectedDataSet = new FlatXmlDataSet(this.getClass()
             .getResource("AsociadoInsertExpectedDataSet.xml"));
        DbUnitDatabaseConnection connection = new
                        DbUnitDatabaseConnection(  dataSource, "public");
        IDataSet actualDataSet = connection
                    .createDataSet(new String[] { "ASOCIADO" });
        DbUnitAssert.assertEqualsDataSet(expectedDataSet,
                                                        actualDataSet);
```

```
	}
	@Test
	@DataSet("AsociadoPersistDataSet.xml")
	public void testUpdate() throws Exception {
		Asociado asociado = new Asociado("10002", "Manuel Alberto",
				"67032134567", "XYZ21");
		asociadoDAO.persist(asociado);
		IDataSet expectedDataSet = new FlatXmlDataSet(this.getClass()
			.getResource("AsociadoUpdateExpectedDataSet.xml"));
		DbUnitDatabaseConnection connection = new
				DbUnitDatabaseConnection(dataSource, "public");
		DbUnitAssert.assertDbContentAsExpected(expectedDataSet,
				connection);
	}
}
```

Fichero fuente de datos AsociadoDataSet.xml.

Fichero AsociadoDataSet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
	<ASOCIADO id="10001" nombre="Ana Laura" ci="88041301234"
codigo="ABC05" />
	<ASOCIADO id="10002" nombre="Miguel Angel" ci="82110956754"
codigo="XYZ21" />
	<ASOCIADO id="10003" nombre="Pedro Pablo" ci="71060690007"
codigo="MNO42" />
	<DIRECCION id_asociado="10001" calle="calle 51" numero="254 A"
localidad="Marianao" />
	<DIRECCION id_asociado="10002" calle="Bolivar" numero="544"
localidad="Madrigal" />
	<DIRECCION id_asociado="10003" calle="Pedrera" numero="101"
localidad="San Miguel" />
</dataset>
```

Fichero fuente de datos AsociadoPersistDataSet.xml

Fichero AsociadoPersistDataSet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
     <ASOCIADO id="10001" nombre="Ana Laura" ci="88041301234"
codigo="ABC05" />
     <ASOCIADO id="10002" nombre="Miguel Angel" ci="82110956754"
codigo="XYZ21" />
     <ASOCIADO id="10003" nombre="Pedro Pablo" ci="71060690007"
codigo="MNO42" />
</dataset>
```

Fichero fuente de datos AsociadoInsertExpectedDataSet.xml

Fichero AsociadoInsertExpectedDataSet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
     <ASOCIADO nombre="Ana Laura" ci="88041301234" codigo="ABC05" />
     <ASOCIADO nombre="Miguel Angel" ci="82110956754" codigo="XYZ21" />
     <ASOCIADO nombre="Manolo Martinez" ci="65121295022" codigo="TRS34" />
     <ASOCIADO nombre="Pedro Pablo" ci="71060690007" codigo="MNO42" />
</dataset>
```

Fichero fuente de datos AsociadoUpdateExpectedDataSet.xml

Fichero AsociadoUpdateExpectedDataSet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
     <ASOCIADO id="10001" nombre="Ana Laura" ci="88041301234"
codigo="ABC05" />
     <ASOCIADO id="10002" nombre="Manuel Alberto" ci="67032134567"
codigo="XYZ21" />
     <ASOCIADO id="10003" nombre="Pedro Pablo" ci="71060690007"
codigo="MNO42" />
</dataset>
```

ANEXO II – PRUEBA A COMPONENTES AISLADOS UTILIZANDO UNITILS Y EASYMOCK.

Prueba de componente aislado usando EasyMock

Clase PeliculaManagerMockTest usando EasyMock

```
package cu.uci.tesis.capitulo3.videoclub.manager;
import junit.framework.Assert;
import org.easymock.classextension.EasyMock;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.unitils.UnitilsJUnit4TestClassRunner;
import org.unitils.easymock.EasyMockUnitils;
import org.unitils.easymock.annotation.Mock;
import org.unitils.inject.annotation.InjectInto;
import org.unitils.inject.annotation.InjectIntoByType;
import org.unitils.inject.annotation.TestedObject;
import cu.uci.tesis.capitulo3.videoclub.dao.PeliculaDAO;
import cu.uci.tesis.capitulo3.videoclub.dominio.GeneroPelicula;
import cu.uci.tesis.capitulo3.videoclub.dominio.PaisProduccion;
import cu.uci.tesis.capitulo3.videoclub.dominio.Pelicula;
import cu.uci.tesis.capitulo3.videoclub.manager.impl.PeliculaManagerImpl;
import cu.uci.tesis.capitulo3.videoclub.validador.PeliculaValidador;

@RunWith(UnitilsJUnit4TestClassRunner.class)
public class PeliculaManagerMockTest {

     @Mock
     @InjectInto(target = "peliculaManager", property = "entidadDAO")
     private PeliculaDAO peliculaDAO;
     @Mock
     @InjectIntoByType
     private PeliculaValidador peliculaValidador;
     @TestedObject
     private PeliculaManager peliculaManager;
```

```
@Before
public void inicializar() {
      peliculaManager = new PeliculaManagerImpl();
}

@Test
public void testActualizarNuevaPelicula() {
      try {
            Pelicula peliculaNew = new Pelicula("10001", "titulo",
                  "director","actor1, actor 2", "sinopsis", 120, 2008,
            new GeneroPelicula("001"), new PaisProduccion("001"));
            Pelicula peliculaOld = new Pelicula("10001");
           peliculaValidador.
                              validarEntidadParaActualizar(peliculaNew);
            EasyMock.expect(peliculaDAO.findById("10001",
                                        false)).andReturn(peliculaOld);
            peliculaValidador.validarEntidadEncontrada(peliculaOld);
            EasyMock.expect(peliculaDAO.persist(peliculaNew))
                                              .andReturn(peliculaNew);
            EasyMockUnitils.replay();
            peliculaManager.modificarDatosPelicula(peliculaNew);
      } catch (Exception error) {
            Assert.fail(error.getMessage());
      }
}

@Test
public void testInsertarNuevaPelicula() {
      try {
            Pelicula pelicula = new Pelicula("titulo", "director",
                        "actor1, actor 2", "sinopsis", 120, 2008,
            new GeneroPelicula("001"), new PaisProduccion("001"));
            peliculaValidador.validarEntidadParaInsertar(pelicula);
            EasyMock.expect(peliculaDAO.persist(pelicula))
                                                  .andReturn(pelicula);
            EasyMockUnitils.replay();
```

```
                peliculaManager.adicionarNuevaPelicula(pelicula);
            } catch (Exception error) {
                Assert.fail(error.getMessage());
            }
        }
}
```

ANEXO III – PRUEBAS DE INTEGRACIÓN UTILIZANDO UNITILS Y DBUNIT.

Prueba de integración de componentes aplicando varias técnicas

Clase VideoClubFachadaTest para probar integración entre componentes

```
package cu.uci.tesis.capitulo3.videoclub.negocio;
import java.util.Calendar;
import java.util.GregorianCalendar;
import javax.sql.DataSource;
import junit.framework.Assert;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.junit.Test;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.unitils.UnitilsJUnit4;
import org.unitils.dbunit.annotation.DataSet;
import org.unitils.dbunit.util.DbUnitAssert;
import org.unitils.dbunit.util.DbUnitDatabaseConnection;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import cu.uci.tesis.capitulo3.nucleo.util.FechaUtils;
import cu.uci.tesis.capitulo3.videoclub.dominio.Prestamo;
public class VideoClubFachadaTest extends UnitilsJUnit4 {
      @SpringBeanByType
      private VideoClubFachada videoClubFachada;
      @SpringBeanByType
      private DataSource dataSource;
      @Test
      @DataSet("PrestamoDataSet.xml")
      public void testDevolverPrestamo() {
            try {
                  Prestamo prestamo = new Prestamo("10001");
                  Calendar calendar = GregorianCalendar.getInstance();
                  calendar.set(Calendar.DATE, 21);
                  calendar.set(Calendar.MONTH, 4);
```

```
                calendar.set(Calendar.YEAR, 2008);
                prestamo.setFechaEntrega(calendar.getTime());
                videoClubFachada.devolverPrestamo(prestamo);
                IDataSet expectedDataSet = new
                                        FlatXmlDataSet(this.getClass().
                            getResource("PrestamoExpectedDataSet.xml"));
                DbUnitDatabaseConnection connection = new
                            DbUnitDatabaseConnection(dataSource, "public");
                DbUnitAssert.
                    assertDbContentAsExpected(expectedDataSet, connection);
          } catch (Exception error) {
                Assert.fail(error.getMessage());
          }
     }
     @SpringApplicationContext
     public final ConfigurableApplicationContext
                                              createApplicationContext() {
          return new
                       ClassPathXmlApplicationContext(getConfigLocations());
     }
     protected String[] getConfigLocations() {
          String[] context = {
              "classpath:cu/uci/tesis/capitulo3/videoclub/configuracion/cu-
                           uci-tesis-capitulo3-videoclub-context-test.xml",
              "classpath:cu/uci/tesis/capitulo3/videoclub/configuracion/cu-
                     uci-tesis-capitulo3-videoclub-dataaccess-context.xml",
              "classpath:cu/uci/tesis/capitulo3/videoclub/configuracion/cu-
                      uci-tesis-capitulo3-videoclub-negocio-context.xml" };
          return context;
     }
}
```

Fichero fuente de datos PrestamoDataSet.xml

Fichero PrestamoDataSet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
```

```
     <PAIS id="001" nombre="Cuba" siglas="CUB" />
     <PAIS id="002" nombre="Venezuela" siglas="VNZ" />
     <GENERO_PELICULA id="001" nombre="Drama" />
     <GENERO_PELICULA id="002" nombre="Comedia" />
     <MODO_CASSETTE id="001" tipo="EP" duracion="360" />

     <ASOCIADO id="10001" nombre="Ana Laura" ci="88041301234"
          codigo="ABC05" />
     <DIRECCION id_asociado="10001" calle="calle 51" numero="254 A"
          localidad="Marianao" />

     <PELICULA id="10001" titulo="tituloA" director="director1"
          actores="actores" sinopsis="sinopsis" duracion="140" anno="2008"
          genero="001" pais="001" />
     <PELICULA id="10002" titulo="tituloB" director="director2"
          actores="actores" sinopsis="sinopsis" duracion="120" anno="1978"
          genero="002" pais="002" />

     <VIDEO_CASSETTE id="10001" modo="001" prestado="true" />

     <CASSETTE_PELICULA id_cassette="10001" id_pelicula="10001" />
     <CASSETTE_PELICULA id_cassette="10001" id_pelicula="10002" />

     <PRESTAMO id="10001" fecha_prestamo="2008-05-01" cantidad_dias="30"
          id_asociado="10001" id_cassette="10001" />


</dataset>
```

Fichero fuente de datos PrestamoExpedtedDataSet.xml

Fichero PrestamoExpedtedDataSet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
     <PAIS id="001" nombre="Cuba" siglas="CUB" />
     <PAIS id="002" nombre="Venezuela" siglas="VNZ" />
```

```
      <GENERO_PELICULA id="001" nombre="Drama" />
      <GENERO_PELICULA id="002" nombre="Comedia" />
      <MODO_CASSETTE id="001" tipo="EP" duracion="360" />

      <ASOCIADO id="10001" nombre="Ana Laura" ci="88041301234"
            codigo="ABC05" />
      <DIRECCION id_asociado="10001" calle="calle 51" numero="254 A"
            localidad="Marianao" />

      <PELICULA id="10001" titulo="tituloA" director="director1"
            actores="actores" sinopsis="sinopsis" duracion="140" anno="2008"
            genero="001" pais="001" />
      <PELICULA id="10002" titulo="tituloB" director="director2"
            actores="actores" sinopsis="sinopsis" duracion="120" anno="1978"
            genero="002" pais="002" />

      <VIDEO_CASSETTE id="10001" modo="001" prestado="false" />

      <CASSETTE_PELICULA id_cassette="10001" id_pelicula="10001" />
      <CASSETTE_PELICULA id_cassette="10001" id_pelicula="10002" />

      <PRESTAMO id="10001" fecha_prestamo="2008-05-01" cantidad_dias="30"
            fecha_entrega="2008-05-21" id_asociado="10001"
id_cassette="10001" />
</dataset>
```

ANEXO IV – CÓDIGO DEL SCRIPT DE LA BASE DE DATOS VIDEOCLUB.

Script de la Base de Datos del caso de estudio Video Club para el gestor PostgreSQL.

Script de la base de datos de prueba

```
CREATE TABLE "PAIS" (id character varying(50) NOT NULL, nombre character
varying(100) NOT NULL, siglas character varying(3) NOT NULL

);

CREATE TABLE "MODO_CASSETTE" (duracion integer NOT NULL, tipo character
varying(5) NOT NULL, id character varying(20) NOT NULL

);

CREATE TABLE "GENERO_PELICULA" (id character varying(20) NOT NULL, nombre
character varying(50) NOT NULL

);

CREATE TABLE "ASOCIADO" (id character varying(20) NOT NULL, nombre
character varying(100) NOT NULL, ci character varying(11) NOT NULL, codigo
character varying(10) NOT NULL

);

CREATE TABLE "PELICULA" (id character varying(20) NOT NULL, titulo
character varying(200) NOT NULL, director character varying(100) NOT NULL,
actores character varying NOT NULL, sinopsis character varying NOT NULL,
duracion integer NOT NULL, anno integer NOT NULL, genero character
varying(20) NOT NULL, pais character varying(20) NOT NULL

);

CREATE TABLE "DIRECCION" (calle character varying(100) NOT NULL, numero
character varying(10) NOT NULL, localidad character varying(100) NOT NULL,
id_asociado character varying(20) NOT NULL

);


CREATE TABLE "VIDEO_CASSETTE" (id character varying(20) NOT NULL, modo
character varying(20) NOT NULL, pretado boolean NOT NULL

);

CREATE TABLE "PRESTAMO" (id character varying(20) NOT NULL, fecha_prestamo
date NOT NULL, cantidad_dias integer NOT NULL, fecha_entrega date,
id_asociado character varying(20) NOT NULL, id_cassette character
varying(20) NOT NULL

);

CREATE TABLE "CASSETTE_PELICULA" (id_cassette character varying(20) NOT
NULL, id_pelicula character varying(20) NOT NULL

);

CREATE SEQUENCE "asociadoSEC" START WITH 10001 INCREMENT BY 1 MAXVALUE
1000000000 MINVALUE 10000 CACHE 1;

CREATE SEQUENCE "cassetteSEC" START WITH 10001 INCREMENT BY 1 MAXVALUE
```

```
1000000000 MINVALUE 10000 CACHE 1;

CREATE SEQUENCE "prestamoSEC" START WITH 10001 INCREMENT BY 1 MAXVALUE
1000000000 MINVALUE 10000 CACHE 1;

CREATE SEQUENCE "peliculaSEC" START WITH 10001 INCREMENT BY 1 MAXVALUE
1000000000 MINVALUE 10000 CACHE 1;
```

GLOSARIO.

A

Aserción - es una verificación donde se produce una falla anticipada y que es emitida con total intención de dar a conocer el motivo por el cual se ha efectuado

C

Cobertura de código - está dada por la medida de cuánta parte del código es probada. Significa inspeccionar si cada línea de código del SBP es ejecutada al menos una vez en alguna prueba.

Código legado – es el código desarrollado por terceros y que no puede ser modificado. Puede ser código fuente, librerías, APIs, configuraciones, etcétera.

J

J2EE – Java Platform Enterprise Edition o Java EE, es una plataforma definida por una especificación y forma parte de la Plataforma Java para desarrollar y ejecutar software de aplicaciones en lenguaje de programación Java con arquitectura de N niveles distribuida, basándose ampliamente en componentes de software modulares y ejecutándose sobre un servidor de aplicaciones.

L

Lector de pruebas - persona que revisa visualmente el código o la configuración de la prueba creada

P

Plugin – un componente enchufable (plug-in en inglés “enchufar") es una aplicación informática que interactúa con otra aplicación para aportarle una función o utilidad específica.

Pruebas cliente - pruebas realizadas por el usuario final del producto para validar su aceptación, ya sea una persona u otro software

Pruebas de aceptación – son realizadas por el usuario final con la intención de criticar y valorar el producto. Verifican el comportamiento completo del sistema o de la aplicación en cuestión.

Pruebas de caja blanca– las pruebas son conscientes del contenido interno de las clases bajo prueba. No sólo prueba lo que es requerido hacer por la clase, sino también cómo lo hace.

Pruebas de caja negra - estas pruebas solo consideran las interfaces públicas de las clases bajo prueba. No se basa en el conocimiento de los detalles de implementación.

Pruebas de componentes - verifican componentes que consisten en grupos de clases que colectivamente proveen algún servicio. Estas se encuentran entre las pruebas unitarias y las pruebas del cliente en cuanto al tamaño del SBP que está siendo verificado.

Pruebas de regresión – intentan descubrir las causas de nuevos errores, carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software, y que fueron inducidos por cambios recientemente realizados en partes de la aplicación que anteriormente no presentaban estos problemas.

Pruebas unitarias - pruebas realizadas a las partes más atómicas del sistema. Verifican el comportamiento de sólo una clase o método, etcétera. que es consecuencia de una decisión de diseño.

R

Refactorización – el término se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento. Su objetivo es el mantenimiento del código que no arregla errores ni añade funcionalidad, sólo el mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto.

S

Sistema Bajo Prueba - es la parte del sistema o el sistema completo en sí que es ejecutado para probar su funcionalidad.

Software sobredetallado - es el software que requiere la implementación de muchas especificaciones que evitan su abstracción en casos más generales.