FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Desarrollo dirigido por pruebas aplicado a Rich Internet Applications
AUTORES: Carlos Sebastián Castañeda
DIRECTOR: Dr. Gustavo Rossi
CODIRECTOR: -

CARRERA: Licenciatura en Informática (plan 2003)

Resumen

La tendencia actual, para el desarrollo de aplicaciones Web, es la utilización de metodologías dirigidas por modelos. Sin embargo, este tipo de aplicaciones requiere de ciclos de desarrollo cortos con un cambio constante en sus requerimientos, y también deben tener un alto grado de usabilidad para satisfacer las necesidades de los usuarios. Por estos motivos, esta tendencia a la que nos referimos se ve limitada ya que este tipo de metodologías utilizan un estilo unificado de desarrollo; en contraposición con el estilo ágil de otros enfoques como la programación extrema XP y el desarrollo dirigido por pruebas TDD. En este trabajo presentamos una metodología que permite combinar el estilo ágil de TDD con una metodología MDWE (Model Driven Web Engineering), para el desarrollo de aplicaciones Web. Luego, propondremos una ampliación de esta metodología para soportar el desarrollo de la segunda generación de aplicaciones Web, mas conocidas como RIA (Rich Internet Applications).

Líneas de Investigación

Aplicaciones Web
Ingeniería Web dirigida por modelos MDWE
Desarrollo dirigido por pruebas TDD
RIA - Rich Internet Applications
Patrones de diseño en aplicaciones Web
Diagramas UID y UI Mockups

Conclusiones

Las aplicaciones RIA plantean un nuevo desafío para los desarrolladores debido a su mayor complejidad en relación a las aplicaciones Web tradicionales. Por este motivo es necesario especificarlas y desarrollarlas a través de alguna metodologías que permita minimizar los errores en cada etapa del desarrollo. El testing de todos los componentes de la aplicación durante todo el ciclo de desarrollo es una tarea fundamental para lograr obtener una aplicación libre de errores y que cumpla los requerimientos iniciales en un ciento por ciento.

Trabajos Realizados

Descripción de la metodología ágil, desarrollo dirigido por pruebas TDD.
Descripción de las aplicaciones RIA y las principales tecnologías para su implementación.
Inclusión de los diagramas UID en el desarrollo Web, e implementación de un editor para dichos diagramas que además permite especificar algunos comportamientos comunes en aplicaciones RIA.
Descripción de un método de desarrollo que combina TDD con las metodologías MDWE.
Descripción de como adaptar la metodología de desarrollo propuesta para soportar el desarrollo de aplicaciones RIA.

Trabajos Futuros

Investigar frameworks que permitan la definición de test de navegación e interacción en las diferentes tecnologías RIA disponibles en el mercado.
Implementar la automatización de la derivación de test de navegación a partir de los diagramas UIDs.
Investigar la ampliación de las metodologías MDWE existentes para soportar el desarrollo de aplicaciones RIA.
Investigar nuevos patrones de diseño RIA, su especificación en los requerimientos del desarrollo y la definición de los test que verifiquen su correcto funcionamiento dentro de la aplicación.

Fecha de la presentación: Marzo 2010

Agradecimientos

En primer lugar quisiera expresar mi agradecimiento a mi familia; a mi mama Mirta quien siempre estuvo a mi lado para darme una mano cada vez que la necesite y a mi hermano Iván con quien en los último años hemos compartido numerosos momentos inolvidables.

A mi director de tesis Gustavo que desde el primer momento me abrió las puertas para comenzar este trabajo. A Esteban cuya ayuda y consejos durante el desarrollo de este trabajo fueron de incalculable valor.

No puedo dejar de agradecer a la familia Caivano (Jorge, Cristina, Estefania y Po), mis amigos del grupo de inglés (Agus, Mari, Ro, Jess, Pocho, Nico y Gaby); mi psicólogo Julio; a Susana, Daniel, Ale, Martín, Diego y a todos aquellos que fueron parte de mi vida en estos años académicos (amigos, compañeros de trabajo y compañeros de estudio).

Por último quiero agradecer a una persona muy especial en mi vida, Mercedes.

A todos les estaré por siempre agradecido.

Este trabajo concluye una etapa de mi vida, y es el comienzo de quien sabe cuantos nuevos desafíos y proyectos.

C. Sebastián Castañeda

Índice general

Capítulo 1

Introducción

La ingeniería dirigida por modelos (MDE de *Model Driven Engineering*) es un metodología de desarrollo de software que se centra en la creación de modelos o abstracciones, mas cercanos a los conceptos de algún dominio en particular que a los conceptos de programación y algoritmos. Este tipo de metodologías se destaca por incrementar la productividad, maximizando la compatibilidad entre sistemas y simplificando el proceso de diseño. A su vez, son menos susceptibles a la introducción de errores debido a la derivación y transformación automática de las entidades que conforman la aplicación.

La ingeniería Web es un dominio especifico en el cual la metodología MDE puede ser aplicada efectivamente. Las metodologías existentes de ingeniería Web basadas en modelos; tales como WebML, UWE, OOHDM, OOWS o OOH, proveen excelentes métodos y herramientas para el diseño y desarrollo de la mayoría de las aplicaciones Web. Esta metodologías abordan diferentes aspectos de una aplicación mediante la utilización de modelos separados (navegación, presentación, flujos de trabajo, etc.) y proveen compiladores de modelos que generan la mayoría de los componentes (lógica y presentación) que conforman una aplicación Web.

Con un enfoque completamente diferente, las métodos ágiles se refieren a un grupo de metodologías de desarrollo de software basadas en el desarrollo iterativo, donde los requerimientos y las soluciones evolucionan a través de la colaboración entre los miembros de equipos interdisciplinarios y auto-organizados. En general, este tipo de metodologías promueven un proceso de administración de proyectos disciplinado que alienta a la inspección frecuente; la adaptación a los cambios; una filosofía de liderazgo que fomenta el trabajo en equipo; la auto-organización y rendición de cuentas; un conjunto de buenas prácticas de ingeniería que permite la rápida construcción y distribución de software de alta calidad; y una forma de encarar los intereses de negocios que permite alinear el desarrollo con las necesidades de los clientes y los intereses de la compañía.

Existen muchas metodologías ágiles de desarrollo de software, entre las cuales podemos nombrar al desarrollo dirigido por pruebas (TDD de *Test-Driven Development*). TDD esta relacionado con los conceptos de *test first programming* de la programación extrema XP. TDD tiene como regla fundamental la creación de test de unidad, que definen los requerimientos, antes de escribir el código funcional en si mismo. TDD utiliza repeticiones de ciclos de desarrollo cortos: en primer lugar el desarrollador escribe un test automático, que fallará en su primer ejecución; este test define la mejora deseada o una nueva funcionalidad; luego se escribe el código para lograr ejecutar el test exitosamente; y finalmente se realiza un *refactoring* del código para que cumpla con determinados estándares (por ejemplo, eliminación de duplicación de código). Según *Kent Beck*, quien ha desarrollado la metodología, TDD promueve el diseño simple e inspira confianza.

La tendencia actual, para el desarrollo de aplicaciones Web, es la utilización de metodologías

MDWE. Sin embargo, no es un tema reciente que el desarrollo de este tipo de aplicaciones requiere de ciclos de desarrollo cortos con un cambio constante en sus requerimientos, y que además deben tener un alto grado de usabilidad para satisfacer las necesidades de los usuarios. Por estos motivos, esta tendencia a la que nos referimos se ve limitada ya que este tipo de metodologías tienden a utilizar un estilo unificado de desarrollo en contraposición con el estilo ágil de otros enfoques como la programación extrema XP y el desarrollo dirigido por pruebas TDD.

En este trabajo abordaremos dos líneas de investigación principales:

- Explicaremos conceptualmente una metodología para el desarrollo de aplicaciones Web que combina el estilo ágil de TDD con el desarrollo formal de MDWE. La idea básica es aplicar los principios de TDD a un enfoque MDWE. De esta manera, los test definidos son ejecutados sobre la aplicación, y en el caso de una falla, se aplican los cambios necesarios a los modelos definidos y no al código generado. Luego, la aplicación podrá ser reconstruida nuevamente a partir de los modelos definidos mediante transformaciones automáticas, y podremos volver a ejecutar los test y continuar con el ciclo de desarrollo. Para la captura de requerimientos utilizaremos los diagramas de interacción de usuario UIDs y definiremos *UI mockups* (bocetos de las interfaces), que permitirán obtener *feedback* sobre el aspecto de la aplicación por parte de los interesados (clientes y usuarios). Estas herramientas servirán como base para definir nuestros modelos y generar los sucesivos prototipos incrementales de la aplicación. Ademas, nos permitirán definir lo que llamaremos test de unidad de navegación (*Navigation unit test*) que extienden el concepto de test de unidad a la navegación real de una aplicación Web. Estos test verificarán que los diagramas UIDs definidos son correctamente representados en la funcionalidad de la aplicación; en primera instancia utilizando los *UI mockups* y luego los sucesivos prototipos de la aplicación generados a partir de los modelos construidos. El proceso general tiene la misma estructura que TDD pero con algunas modificaciones y agregados para permitir la compatibilidad de ambas metodologías. Las principales diferencias con TDD convencional son: la utilización de modelos (negocio, navegación y presentación) en vez de la escritura directa de código, la captura de requerimientos mediante diagramas UIDs y *UI mockups*, y la definición de test que controlen la evolución de los modelos durante el todo el ciclo de desarrollo. Estos test validan los requerimientos funcionales y de usabilidad y tienen en cuenta los *refactorings* que puede sufrir la aplicación durante su desarrollo.
- Describiremos los puntos a tener en cuenta para adaptar la metodología propuesta para ser utilizada en el desarrollo de aplicaciones Web enriquecidas (RIA de *Rich Internet Applications*). Este tipo de aplicaciones son la segunda generación de aplicaciones Web y presentan nuevos desafíos a los desarrolladores. Como parte de la ampliación de la metodología describiremos una forma de extender los test de unidad de navegación para que puedan verificar algunos de los patrones de diseño que son comunes en este tipo de aplicaciones.

1.1. Organización del documento

El resto de este trabajo se estructura de la siguiente manera:

- **Capítulo 2**: *Desarrollo dirigido por pruebas* - Describiremos brevemente las metodologías ágiles de la programación extrema XP y el desarrollo dirigido por pruebas TDD para el cual explicaremos los pasos que conforman su ciclo de desarrollo.
- **Capítulo 3**: *Rich Internet Applications* - Definiremos las principales características de las

aplicaciones RIA y su contraposición con las aplicaciones Web convencionales. Describiremos brevemente algunos de los principales frameworks para su desarrollo.

- **Capítulo 4**: *User Interaction Diagrams* - Describiremos la notación de diagramas UIDs y su rol en el desarrollo de aplicaciones Web.
- **Capítulo 5**: *MDWE y TDD en el desarrollo de aplicaciones Web* - Explicaremos una metodología que permite combinar las metodologías ágiles, en particular TDD, con las metodologías MDWE. Definiremos los conceptos de test de navegación y explicaremos como capturar los requerimientos mediante los diagramas UIDs y los *UI Mockups*. Para ilustrar las ideas utilizaremos el lenguaje de modelado *WebML*, la herramienta *WebRatio* (implementación de WebML) y el *framework* para definición de test *Selenium*.
- **Capítulo 6**: *MDWE y TDD en el desarrollo de aplicaciones RIA* - Definiremos los puntos a tener en cuenta para poder adaptar nuestra metodología para ser utilizada en el desarrollo de aplicaciones RIA. Hay varios puntos a tener en cuenta: la tecnología RIA utilizada para la aplicación resultante del desarrollo, la utilización de una metodología MDWE que permita modelar aspectos propios de este tipo de aplicaciones y la disponibilidad de un framework para la definición de test de navegación que pueda interactuar con la tecnología RIA utilizada.
- **Capítulo 7**: *Definición de tests para requerimientos RIA* - Como parte de la ampliación de nuestra metodología explicaremos como podemos incluir en los test de navegación la verificación de varios comportamientos comunes (patrones de diseño) en aplicaciones RIA. Aunque las ideas expuestas en este capitulo son de carácter general, como prueba de concepto utilizaremos estos test sobre aplicaciones RIA híbridas (aplicaciones Web convencionales con componentes RIA) y nos basaremos principalmente en algunos de los patrones descriptos en la librería *Yahoo! Design Patterns* [YDPL09].
- **Capítulo 8**: *Conclusiones, críticas y trabajos futuros* - Mis conclusiones y aportes a nivel personal; las criticas de los desarrollos realizados; y las líneas de investigación y trabajos a futuro que podrían derivarse de este trabajo.
- **Apéndice I**: *Notación UID* - Especificación completa de la notación para diagramas UIDs.
- **Apéndice II**: *Modelado RIA en WebML* - Explicación de una posible alternativa para ampliar el lenguaje *WebML* para permitir el modelado de características y funcionalidad RIA.

Capítulo 2

Desarrollo dirigido por pruebas

El rápido crecimiento en popularidad de la programación extrema (XP) ha posicionado las prácticas de testing de software como parte fundamental del desarrollo de aplicaciones. En un primer momento, el testing de aplicaciones era una práctica muy descuidada y realizada a último momento lo cual ocasionaba que muchos proyectos no pudieran ser terminados en tiempo y forma. Sin embargo, en la actualidad las expectativas de obtener productos de alta calidad, hacen que el testing se haya convertido en un componente clave del proceso de desarrollo.

XP acelera el testing del software mediante su integración completa en el desarrollo. Esto ha obligado a los profesionales del software a repensar su posición con respecto a la tareas de testing. XP fuerza al equipo completo de desarrollo a llevar a cabo tareas de testing. De hecho, es una actividad tan critica en la metodología XP que los programadores están obligados a escribir test automáticos (unit test) antes de comenzar con las tareas de codificación (test first programming).

El desarrollo dirigido por pruebas (TDD de Test-Driven Development) esta relacionado con los conceptos de Test-First Programming de la programación extrema, allá por el año 1999, pero recientemente se ha creado un gran interés es esta metodología por si misma.

El desarrollo dirigido por pruebas (TDD) es una aproximación evolutiva para desarrollar software que combina la realización de casos de prueba, donde se escribe un test antes de escribir el código de producción (suficiente para completar y ejecutar exitosamente ese test) para luego refactorizar el código obtenido a fin de mejorar su calidad (por ejemplo, eliminar duplicación de código). El objetivo primario de TDD es la especificación y no la validación. En otras palabras, es una manera de pensar mediante el diseño antes de escribir el código funcional. Desde otro punto de vista, TDD es una técnica de programación cuyo objetivo es escribir código limpio y claro que funcione correctamente.

En este capitulo describiremos brevemente los conceptos de programación extrema y su relación con las tareas de testing y QA (*Quality Assurance*); así como también la metodología de desarrollo dirigida por pruebas (TDD). En [Ast03], [Beck03] y [CrHo03] podemos encontrar una descripción detallada sobre estas metodologías, también encontraremos varios ejemplos concretos de aplicación de TDD.

2.1. La Programación Extrema

La programación extrema o XP (de *Extreme Programming*), fue introducida por primera vez por *Kent Beck* en 1996. Desde entonces, una gran cantidad de programadores han adoptado la técnica con gran entusiasmo.

XP nos es una técnica donde todo es válido y no hay reglas precisas. En realidad, es un enfoque bien disciplinado orientado al desarrollo de software. Y a pesar de lo que pueda inducir su nombre, los practicantes de XP son probablemente mas conscientes acerca de respetar ciertas reglas que los desarrolladores que utilizan metodologías tradicionales. Solo existen algunas pocas reglas en XP y están orientadas a un tipo de proceso de desarrollo completamente diferente.

La mejor forma de describir y resumir a XP es a través del conjunto de valores y prácticas que promueve:

> La programación extrema es una disciplina de desarrollo de software basada en sus valores de simplicidad, comunicación, realimentación y coraje. Funciona uniendo al equipo completo en presencia de prácticas simples, con suficiente coraje para permitir al equipo tener en claro donde se encuentran y armonizar las prácticas para su situación particular.

Los cuatro valores de la programación extrema son la comunicación, simplicidad, realimentación y el coraje. Estos valores proviene de una realización en donde se afirma que muchos de los problemas que afectan al desarrollo de software pueden atribuirse a fallas en uno de estos cuatro ámbitos y, en consecuencia, que las mejoras en estas áreas darían lugar a un proceso de desarrollo significativamente mejor. Los cuatro valores conllevan a un conjunto de 12 prácticas, que esencialmente conforman las reglas y fundamentos de las programación extrema. Cabe señalar que esta lista relativamente breve tiene como objetivo estar presente en la memoria de los programadores, teniendo como consecuencia que los practicantes de XP sean muy disciplinados al seguir las reglas.

2.1.1. Las 12 prácticas de la programación extrema

A continuación describiremos las 12 prácticas de la programación extrema, las mismas se encuentran agrupadas bajo los valores de las que son principalmente derivadas.

Comunicación

- `Clientes in situ` - El equipo de desarrollo tiene acceso e interacción continua con los clientes, quienes en realidad son parte del equipo del proyecto. Los clientes describen la funcionalidad del sistema en *user stories*, negocian el tiempo y planificación de los *releases*, y toman todas las decisiones que afectan los objetivos e intereses del negocio. Siempre están disponibles para contestar preguntas sobre detalles de las *user stories* durante las tareas de programación. Ayudan con el testing funcional y toman las decisiones finales de permitir al sistema continuar en producción o ser detenido.

- `Programación en pares` - Se conforman equipos de dos programadores para escribir todo el código de producción. Esto significa que al menos dos personas están siempre íntimamente familiarizadas con cada parte del sistema, y cada línea de código es revisada en el mismo instante que es escrita.

- `Estándares de codificación` - Todos lo miembros del equipo codifican utilizando los mismo estándares de codificación. Esto permite mantener el código consistente y que carezca de peculiaridades idiosincrásicas que podrían complicar el entendimiento y *refactoring* por parte del resto del equipo.

Simplicidad

- `Metáforas` - Cada proyecto tiene un metáfora organizativa de la cual se derivan los nombres de las entidades de programación (por ejemplo, clases y métodos), proveyendo una manera simple de recordar las convenciones de nombres.

- `Diseño simple` - Los desarrolladores XP siempre realizan la implementación mas simple posible que cubra el requerimiento de turno. Nunca resuelven un problema mas general que el problema especifico del momento, y nunca agregan funcionalidad antes que sea necesaria.
- `Refactoring` - La duplicación y complejidad innecesaria es removida del código durante cada iteración, aun cuando esto implique modificar componentes que ya están terminados. Los test de unidad automatizados son utilizados para verificar que cada cambio realizado no corrompe la funcionalidad agregada y/o modificada.

Realimentación

- `Testing` - Los test de unidad son el núcleo de XP. Cada pieza de código tiene un conjunto de test automatizados, que son mantenidos junto con el resto de los componentes en el repositorio de código. Los programadores escriben tests de unidad antes de escribir el código funcional. Ninguna modificación o *refactoring* esta completo hasta que la totalidad de los test de unidad se ejecuten exitosamente. Los test de aceptación validan grandes bloques de funcionalidad del sistema, tales como las *user stories*. Cuando todos los test de aceptación son correctos para una *user story* particular, esa *story* se considera completa.
- `Integración continua` - Los agregados y modificaciones del código son integrados al sistema al menos diariamente y los test de unidad deben ser ejecutados exitosamente antes y después de cada integración.
- `Pequeños releases` - El conjunto de características mas útiles y prioritarias es identificado para el primer *release*, y cada nuevo release es realizado tan pronto como sea posible, con unas pocas características adicionales en cada nueva versión.

Coraje

- `Planificación` - Los clientes y desarrolladores cooperan en planificar el orden y tiempo estimado de implementación de las características y funcionalidades del sistema en cada *release*. Los clientes proveen las especificaciones del sistema en la forma de *user stories*, y los desarrolladores proveen las estimaciones de los tiempos y esfuerzos requeridos para completar cada requerimiento. Los clientes toman las decisiones sobre que *stories* implementar y en que orden. Los desarrolladores y clientes negocian en manera conjunta la secuencia de iteraciones, cada una de estas incluye un conjunto de *stories* que culminan con el *release* final, el cual contiene a todas las *stories* definidas en primer instancia.
- `Código de propiedad colectiva` - Ninguna persona es propietaria de los módulos que componen el sistema de software. Se espera que todos los programadores estén en condiciones de trabajar y refactorizar cualquier parte del sistema en cualquier momento.
- `Ritmo sostenible` - Los miembros del equipo deben estar en condiciones de trabajar a un ritmo sostenido por un largo periodo de tiempo sin tener que poner en riesgo su salud física y mental, familia o productividad. Ocasionalmente, pueden existir picos de trabajo excesivo, pero si esto se prolonga por varias semanas es un signo de un problema mas profundo en la administración del proyecto.

2.1.2. Como resuelve XP los problemas de Testing y QA

En pocas palabras XP es una colección de practicas simples, llevadas al extremo. Tal vez la cosa mas extrema de todos los equipos que utilizan XP es la disciplina con la que siguen las prácticas.

Muchas de las prácticas especificadas por XP están muy relacionadas a aspectos de testing y aseguramiento de calidad (QA). A continuación se describen tres de los problemas, en estos aspectos, que XP puede resolver:

Los recursos de testing de sistema y aceptación son malgastados en errores de unidad e integración Uno de los problemas mas comunes que los testers experimentan en los proyectos de software tradicionales es ejecutar tests a nivel de sistema sobre código que nunca fue testeado a niveles de unidad e integración. Esta es una forma costosa e ineficiente de encontrar errores a nivel de unidad e integración y pueden fácilmente consumir todo el tiempo disponible para los testing de sistema y aceptación. Puede suceder que el sistema sea entregado fuera de termino o que los testing de sistema y aceptación nunca puedan ser completados, y que el software sufra significativos problemas de calidad en operación.

Esta situación es especialmente frustrante para los testers, porque no pueden hacer demasiado para resolver el problema. No importa que tan eficientes y/o automatizados sean los test que desarrollan, ya que no pueden agregar funcionalidad o calidad al sistema, solo pueden remitirse a detectar su presencia y/o ausencia.

XP evita este escenario a través de la automatización total de los test de unidad e integración continua. Las parejas de programadores detectan y corrigen los errores de unidad e integración durante las sesiones de codificación. Para el momento en que el código es verificado con los test de aceptación realiza lo que los programadores en realidad deseaban y los test de aceptación pueden enfocarse en que porcentaje el sistema cumple con las expectativas de los clientes.

Requerimientos omitidos o fuera de termino Otro problema común, en el desarrollo de los test, sucede cuando los requerimientos son omitidos o se especifican en momentos posteriores a la fase de captura de requerimientos. Los extensos y detallados documentos de requerimientos de los desarrollos de software tradicionales son ideales para utilizar en el desarrollo de los test pero frecuentemente son muy costosos de crear y mantener, y dejan al tester sin alternativas cuando inevitablemente se tornan obsoletos. Esto ocasiona que los test desarrollados para los requerimientos surgidos fuera de termino sean inexactos.

XP resuelve este problema reemplazando los documentos de requerimientos extensos con pequeñas *user stories* que pueden ser fácilmente mantenidas y ampliadas por los clientes pertenecientes al proyecto, cuya función es servir de consulta para todos los detalles que las *stories* no contemplan.

Diferencias entre el sistema y las expectativas del los usuarios Por último, uno de las cosas mas problemáticas, para los tester o personal de QA, es cuando hay una diferencia significativa entre lo que el sistema hace y lo que los usuarios esperan. Esto es algo que sucede con facilidad en los desarrollos de software tradicionales, cuando el cliente esta fuera del ciclo de desarrollo luego de la fase de requerimientos, y pueden pasar meses e incluso años antes que se distribuya alguna funcionalidad. Haciendo énfasis en estas diferencias los testers pueden ser vistos como si tuvieran una visión negativa, y no fueran parte del equipo, y eventualmente como *el enemigo*. Si no se hace énfasis en estas diferencias puede causar que los clientes no estén satisfechos y que se refleje este malestar en contra del equipo de testing (por ejemplo, porque este problema no fue detectado por el área de testing?).

XP elimina este escenario produciendo frecuentes y pequeños *releases* que proveen exactamente aquellas características que el cliente prioriza como las mas importantes. Si se genera una diferencia de expectativas, se vuelve naturalmente visible en el espacio de una iteración (a lo sumo dos a tres semanas) y es abordada por el equipo completo.

Estos tres problemas son en general síntomas de problemas mas profundos dentro de un proyecto de desarrollo de software. Si bien XP no es la solución para las causas de estos problemas, ayudan a minimizarlos y mejoran calidad de vida de los profesionales del testing y QA.

2.2. Desarrollo dirigido por pruebas: Definición

Para entender mejor el concepto del desarrollo dirigido por pruebas (TDD de *Test Driven Development*), primero debemos comprender el concepto de *Test First Design*(TFD). La figura 2.1 muestra los pasos involucrados en la técnica de TFD. El primer paso es agregar un nuevo test: básicamente suficiente código para producir una falla. El siguiente paso es ejecutar los test: para asegurar que el nuevo test realmente produce una falla. Luego se actualiza el código funcional: para lograr que el nuevo test tenga un resultado exitoso. El cuarto paso es volver a ejecutar los test nuevamente: si se produce una falla se necesita actualizar el código funcional y retestear. Una vez que los test son exitosos el siguiente paso es volver a comenzar un nuevo ciclo (posiblemente se necesite una refactorización del código para eliminar duplicaciones del diseño, convirtiendo TFD en TDD).

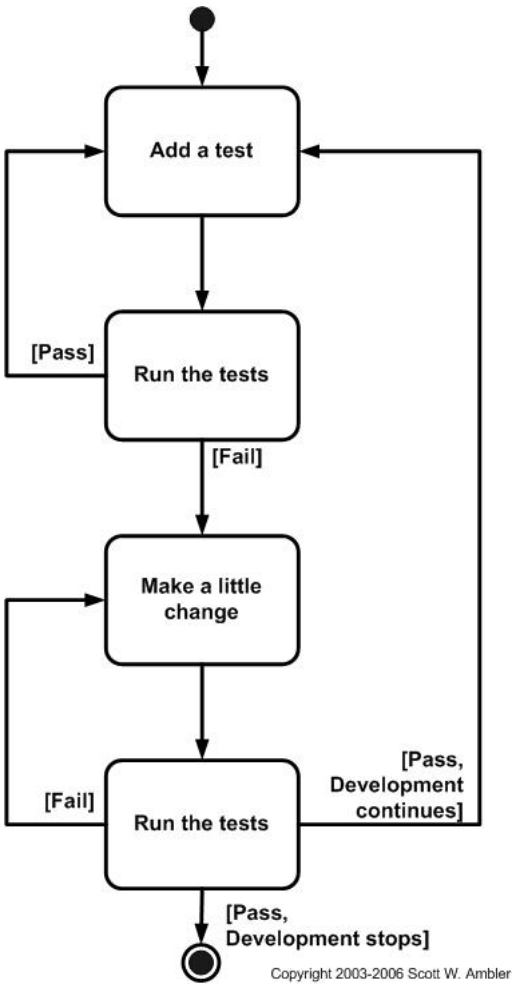

Figura 2.1: `Pasos de test first design (TFD)`

TDD se puede describir con la siguiente formula:

$$TDD = Refactoring + TFD.$$

TDD redefine completamente el desarrollo tradicional. Cuando se implementa una nueva característica, la primer pregunta que uno debería realizarse es si el diseño existente es el mejor diseño posible que permite implementar la funcionalidad. Si es así, se procede mediante TFD. Sino, se realiza una refactorización local para cambiar la porción del diseño afectada por la nueva característica, permitiendo agregarla tan simple como sea posible. Como resultado siempre se estará mejorando la calidad del diseño, haciendo mas fácil trabajar con él en el futuro.

En vez de escribir el código funcional y luego el código de testing como última etapa, si es que llega a realizarse, se puede escribir los test antes que código el funcional. Mas aun, esto se realiza en pequeños pasos: un paso de test y un pequeño fragmento de código funcional a la vez. Un programador que utiliza TDD debería reusarse a escribir una nueva función hasta que exista un test que falle porque la función no esta presente o implementada. Incluso debería negarse a escribir una sola línea de código hasta que exista un test para la misma. Una vez que el test esta en su lugar, el programador realiza el trabajo requerido para asegurar que la test suite se ejecuta exitosamente (el nuevo código podría producir que varios de los test ya existentes dejen de funcionar, ademas del nuevo). Esto suena simple en principio, pero cuando se aprende TDD por primera vez, requiere de una gran disciplina, puesto que es muy común estar tentado a escribir código funcional en primer termino sin escribir un nuevo test. Una de las ventajas de la programación en pares es facilitar que no se olviden las pautas de desarrollo.

La principal asunción de TDD es que se cuenta con un *framework* para testing de unidad disponible. Los desarrolladores de software ágiles en general utilizan la familia *xUnit* de herramientas *open source*, tales comos *JUnit* o *PyUnit*, aunque también existen otras opciones comerciales. Sin estas herramientas TDD es virtualmente imposible.

2.3. Ciclo del desarrollo dirigido por pruebas

La figura 2.2 muestra una representación del ciclo de desarrollo de TDD. Cada iteración de TDD se compone de cinco pasos:

1. Agregar un nuevo test;
2. Reejecutar todos los tests y verificar que el nuevo falla;
3. Escribir la lógica necesaria para ejecutar con éxito el nuevo test;
4. Reejecutar todos los tests y comprobar que todos se ejecutan correctamente;
5. Refactorizar el código para eliminar ineficiencias (por ejemplo duplicación de código).

1. **Agregar un nuevo test** - En TDD, cada nueva funcionalidad comienza con el desarrollo y escritura de un test. Este nuevo test debe inevitablemente fallar porque es escrito *antes* que la funcionalidad haya sido implementada (si no falla, entonces la nueva funcionalidad propuesta es omitida). Para escribir un test, el desarrollador debe entender claramente la especificación y los requerimientos. La forma de interiorizarse y tener claros los requerimientos es mediante casos de uso y *user stories* que cubran los requerimientos y condiciones excepcionales. Esto puede también implicar una variante, o modificación de un test ya existente. Esta es una característica que diferencia a TDD de la escritura de tests de unidad luego de que el código

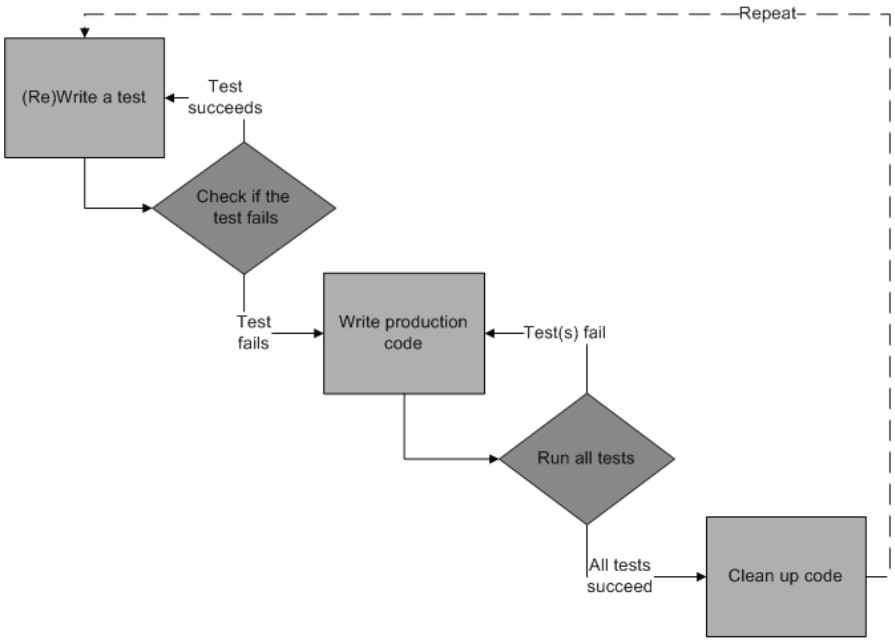

Figura 2.2: `Ciclo de TDD`

esta escrito ya que obliga al desarrollador a enfocarse en los requerimientos antes de escribir el código, una sutil pero importante diferencia.

2. **Ejecutar todos los test** - Esta acción valida que la red de test continúe funcionado correctamente y que el nuevo test no resulta exitoso por error, sin requerir agregar nuevo código. Este paso también verifica al test en si mismo; en el peor caso: existe la posibilidad que el nuevo test siempre sea exitoso, y por ende resulte de poco valor. El nuevo test debería fallar por la razón esperada. Esto incrementa la confianza (aunque no es una garantía al ciento por ciento) de que se esta testeando la funcionalidad correcta.

3. **Escribir código funcional para el nuevo test** - El siguiente paso es escribir el código que produzca que la ejecución del nuevo test sea exitoso. El nuevo código en esta etapa puede no ser el optimo, por ejemplo permite la ejecución exitosa del test en una manera poco elegante. Esto es aceptable porque en el paso siguiente del ciclo se mejorará la implementación del nuevo código. Es importante que el código sea diseñado con el único objetivo de que el resultado de ejecutar el nuevo test sea exitoso; ninguna otra funcionalidad debería ser deducida o permitida en esta etapa.

4. **Reejecutar todos los test** - Si todos los casos de test son exitosos, el programador puede confiar en que el código satisface todos los requerimientos testeados. Este es un buen punto para comenzar la etapa final del ciclo.

5. **Refactoring del código** - En esta etapa el código es optimizado y mejorado tanto como sea posible y necesario. Re-ejecutando los casos de test, el desarrollador puede confiar en que el *refactoring* no esta anulando la funcionalidad existente. El concepto de remover duplicaciones es un aspecto importante de cualquier diseño de software. En este caso, también se aplica para remover la duplicación entre el código de los test y de producción, por ejemplo *magic numbers* o *strings* que han sido replicados en ambos, con el objetivo de ejecutar exitosamente el test durante la etapa 3.

6. **Repetir el ciclo** - Comenzando con un nuevo test, se repite todo el ciclo para agregar nueva funcionalidad. El tamaño de los pasos debe ser siempre pequeño (de 1 a 10 ediciones entre cada ejecución de test). Si el nuevo código no satisface rápidamente el nuevo test, y otros test fallan inesperadamente, el programador debería deshacer o revertir los cambios en vez de optar por un *debugging* excesivo. La integración continua ayuda a proveer puntos de verificación reversibles. Cuando se utiliza una librería externa es importante no realizar incrementos que sean tan pequeños que efectivamente solo verifiquen la librería en si misma, a menos que haya una razón para creer que dicha librería contiene errores o no es lo suficientemente completa para cubrir todas las necesidades del programa principal que esta siendo escrito.

2.4. Resumen

La programación extrema o XP es una metodología para el desarrollo de software con un enfoque completamente diferente a las metodologías de desarrollo de software tradicionales. XP se basa en sus valores de simplicidad, comunicación, realimentación y coraje y define 12 prácticas:

- Clientes in situ
- Programación en pares
- Estándares de codificación
- Metáforas de proyecto
- Diseño simple
- Refactoring
- Testing
- Integración continua
- Pequeños releases
- Planificación
- Código de propiedad colectiva
- Ritmo sostenible

Estas prácticas pueden ser fácilmente recordadas por el equipo de desarrollo y en consecuencia ser muy disciplinados al momento de aplicarlas. La programación extrema minimiza varios problemas que son comunes en las metodologías de desarrollo de software tradicionales. En particular, evita que los recursos de testing de sistema y aceptación sean malgastados en errores de unidad e integración, permite verificar en tiempo y forma el correcto funcionamiento de los requerimientos omitidos o agregados posteriormente a la fase de requerimientos, y minimiza las brechas que pueden existir entre la funcionalidad que provee el sistema y las expectativas de los usuarios.

TDD es una metodología ágil que surgió en primera instancia junto con los conceptos de programación extrema, pero que ha tomado gran interés por si mismo. Se basa en las ideas de *test first design*, agregando el concepto de refactoring luego de terminado cada ciclo de codificación. Cada iteración de TDD se compone de cinco pasos: 1) agregar un nuevo test, 2) reejecutar todos los tests y verificar que el nuevo falla, 3) escribir la lógica necesaria para ejecutar con éxito el nuevo

test, 4) reejecutar todos los tests y comprobar que todos son exitosos y 5) refactorizar el código para eliminar ineficiencias.

Para poner en práctica esta metodología es fundamental tener a disposición un framework para testing de unidad. La familia de frameworks *xUnit* es la mas popular y tiene su correspondiente implementación en la mayoría de los lenguajes de programación mas utilizados. Sin un framework de estas características, TDD es virtualmente imposible de aplicar.

Capítulo 3

Rich Internet Applications

En los últimos años la Web se ha convertido en la plataforma de referencia para el desarrollo de soluciones de negocios integradas. Debido al creciente aumento en la complejidad de estas aplicaciones, las tecnologías Web predominantes desde hace algunos años comenzaron a mostrar limitaciones en su usabilidad e interactividad. La experiencia de los usuarios en las aplicaciones Web tradicionales no es comparable a las interfaces de las aplicaciones de escritorio, la respuesta es mas lenta debido a la sobrecarga impuesta por la red y los innecesarios requerimientos de acceso al servidor, y la falta de soporte para su utilización fuera de línea.

Las aplicaciones RIA han sido propuestas como la respuesta a estos problemas. Estas aplicaciones proveen interfaces sofisticadas para representar información y procesos complejos, mientras que a su vez minimizan las transferencias de datos cliente-servidor y trasladan las capas de interacción y presentación desde el servidor al cliente.

En este capitulo describiremos las principales características de una aplicación RIA y sus diferencias con una aplicación Web tradicional. Por último, describiremos brevemente algunas de las tecnologías para el desarrollo de aplicaciones RIA mas populares de la actualidad.

3.1. Orígenes de las aplicaciones RIA

En sus inicios Internet se utilizaba para publicar y compartir datos estáticos en HTML. Al poco tiempo se dio paso a las aplicaciones de negocios que demandaban información mas dinámica. Estos requerimientos fueron soportados mediante *scripts* como *Javascript* que mejoraban las experiencia de los usuarios. Sin embargo, el surgimiento de sistemas de negocios mas complejos presento nuevos retos para la Web. En vez de aplicaciones simples, los nuevos sistemas de negocios requerían características que involucraban numerosos pasos para completar las transacciones resultando en una confusión para los usuarios, ya que se necesitaban varios pasos para realizar una única transacción que involucraba pasar por múltiples formularios e interfaces a través de la aplicación. Esto, sumado a la naturaleza sincrónica de estas aplicaciones, reducía las prestaciones generales de las mismas. Con las aplicaciones RIA no solo se solucionan estos inconvenientes mediante la usabilidad de las tradicionales aplicaciones de escritorio sino que también se incorporan características que servirán para cubrir los requerimientos de negocios, cada vez mas complejos.

El termino *Rich Internet Application* fue introducido por Macromedia en 2002 [Allai02], para describir la unificación de las tradicionales aplicaciones de escritorio y las aplicaciones Web, con el objetivo de aprovechar las ventajas y superar los inconvenientes de ambas arquitecturas. Sin embargo, han ganado su popularidad recientemente a causa de varios factores entre los cuales

podemos mencionar:

1. `Ancho de banda` - La mayoría de las aplicaciones RIA requieren la transferencia de una aplicación cliente inicial. Este requerimiento presentaba un gran obstáculo cuando los usuarios de Internet solo disponían de conexiones *dial-up*. Con el aumento del ancho de banda es posible transportar datos mas grandes, tales como audio y vídeo.

2. `Capacidad de computo` - La diferencia de capacidad de computo entre las computadoras personales y los servidores se a reducido significativamente. Por ejemplo, las computadoras de escritorio, *laptops* y *PDAs* ahora ofrecen audio y vídeo. RIA utiliza mas poder de computo del lado del cliente que las tradicionales aplicaciones Web (*send and receive*). La reducción de esta brecha a creado un entorno ideal para el desarrollo de las aplicaciones RIA.

3. `Mejor respuesta a las acciones del usuario` - Hoy mas que nunca, los negocios demandan aplicaciones mas complejas y con rápida respuesta. En general, se requieren de interfaces de usuario muy sofisticadas para presentar la información y responder rápidamente a las acciones de los usuarios. Se vuelve extremadamente dificultoso para las aplicaciones basadas en HTML cubrir este tipo de requerimientos.

4. `Las influencia de las companías lideres` - Las grandes companías han comenzado a ver los beneficios de las aplicaciones RIA. Por ejemplo, *Google Maps* y *Gmail* utilizan RIA mediante el uso de la técnica llamada AJAX (*Asynchronous JavaScript and XML*). *Macromedia* abrió las puertas a muchas companías y desarrolladores cuando lanzo al mercado su tecnología *Flex* en 2004. *Microsoft .NET* comparte la misma visión ya que promueve el desarrollo de aplicaciones cliente complejas, mientras que a su vez reduce los tiempos y costos de desarrollo. También hay que destacar las alternativas *open source* para el desarrollo RIA. *Laszlo Systems* distribuye la plataforma servidor *Open Laszlo* gratuitamente mientras que *Mozilla Foundation* ofrece una tecnología RIA propietaria llamada XUL (XML User Interface Language).

5. `Web Services y SOA` - Desde la perspectiva del desarrollador, el surgimiento de los servicios Web (*Web services*) y las arquitecturas orientadas a servicios (SOA) cambiaron la manera en que se desarrollan las aplicaciones de negocios (*middleware*). Su aparición tuvo una fuerte influencia sobre como deberían ser desarrolladas las capas de presentación y la comunicación con las capas de servicio. Los Web services y SOA permiten a las capas de presentación ser completamente independientes de las capas de lógica de negocios. La mayoría de las plataformas RIA son capas de presentación que funcionan como un *plugin* sobre el *middleware* existente, gracias a estas dos componentes. No se puede dejar de mencionar a XML, que es la base no solo para las aplicaciones RIA sino también para los Web services y las arquitecturas orientadas a servicios.

3.2. Aplicaciones web y aplicaciones de escritorio

Para entender mas claramente las características de una aplicación RIA, es necesario enumerar las ventajas que otorgan tanto las aplicaciones basadas en un navegador como las aplicaciones de escritorio tradicionales.

Las aplicaciones basadas en un navegador Web, son usualmente preferidas por las siguientes razones:

- Los *scripts/tags* estándar son fáciles de desarrollar (desarrollo rápido y de bajo costo)

- No se necesita instalación, actualizaciones o parches (bajo costos de distribución y mantenimiento)
- Las aplicaciones son accesibles desde redes de computadores (disponibilidad y flexibilidad)
- Las aplicaciones pueden ejecutarse en diferentes sistemas operativos (independencia de la plataforma)
- Las interfaces de usuario (UI) son simples y estandarizadas (curva de aprendizaje baja para los usuarios finales)

Por su parte, las aplicaciones de escritorio tienen las siguientes características destacables:

- Experiencia de los usuarios mas enriquecida (audio , vídeo, comunicaciones, etc.)
- No tienen una recarga constante de paginas y cambios de contexto (continuidad visual)
- Soporte para trabajo en línea y fuera de línea
- Permite el desarrollo de aplicaciones mas complejas (ejemplo, MS Outlook vs Webmail)
- Tienen mayor respuesta e interactividad

Las aplicaciones RIA capitalizan los puntos fuertes de ambos tipos de aplicaciones. RIA es un conjunto de tecnologías que permite el desarrollo de algunas o todas las características citadas anteriormente. La tabla 3.1 muestra una comparación de las características de las aplicaciones de escritorio (incluyendo cliente-servidor) y las aplicaciones Web convencionales HTML+HTTP, resaltando el potencial de las aplicaciones RIA.

Característica	**Escritorio C/S**	**Web**	**RIA**
Cliente universal (navegador)	NO	SI	SI
Instalación en cliente	Compleja	Simple	Simple
Capacidades de interacción	Alta	Limitada	Alta
Lógica de negocios en servidor	SI	SI	SI
Lógica de negocios en cliente	SI	Limitada	SI
Actualización completa de páginas	NO	SI	NO
Requerimientos al servidor frecuentes	NO	SI	NO
Comunicación servidor-cliente	SI	NO	SI
Funcionalidad fuera de línea	SI	NO	SI

Cuadro 3.1: `Comparación entre aplicaciones de escritorio, Web y RIA`

3.3. Características de las aplicaciones RIA

Son varias las características y beneficios que distinguen a una aplicación RIA, entre ellas podemos enumerar las siguientes:

- `Accesibilidad` - Aunque las aplicaciones RIA brindan una nueva forma de interactuar con los usuarios, retienen sus capacidades de fácil accesos desde virtualmente cualquier lugar en el mundo.

- `Continuidad visual` - La información en una aplicación RIA esta siempre visible a los usuarios eliminando las re-actualizaciones constantes de páginas y los cambios de contexto.
- `Reducción del trafico de red` - Ya que las aplicaciones trabajan en un modo asincrónico y la mayor parte del procesamiento es intensivo del lado del cliente, se reduce el trafico de red resultando en una mejora de las prestaciones generales del sistema y las experiencias para los usuarios.
- `Interfaces enriquecidas y mas interactivas` - RIA retiene las características de la Web para presentar información compleja a los usuarios pero aumenta estas capacidades incorporando interfaces enriquecidas y mayor interactividad. Las aplicaciones RIA soportan un conjunto amplio de elementos visuales como imágenes, gráficos, vídeo, etc. que se integran sin problemas con el núcleo de los elementos de negocio y permiten una interactividad y experiencia mas rica para los usuarios.
- `Tiempo real` - Las aplicaciones RIA trabajan en tiempo real permitiendo el desarrollo de servicios de negocios como mensajeros de conferencias y servicios bajo demanda.
- `Facilidad de uso` - Las aplicaciones RIA permiten a los usuarios entender y utilizar aplicaciones de negocios complejas reduciendo la transversalidad de la Web y mostrando la mayor parte de la información en una única interfaz integrada sin la necesidad de moverse desde un contexto a otro.

3.4. Arquitectura de las aplicaciones RIA

La característica distintiva de la arquitectura RIA de las arquitecturas Web tradicionales es su motor de renderización. RIA utiliza un motor de renderización que trabaja desde el lado del cliente y actúa como un intermediario entre el servidor Web y el usuario.

Del lado del servidor, la mayoría de las aplicaciones RIA incluyen un componente que trabaja para el motor de renderización procesando la lógica de negocios requerida del lado del cliente.

Como resultado de esta arquitectura, RIA ofrece las siguientes ventajas:

- La información es obtenida desde el servidor y actualizada en un modo asincrónico según las necesidades del usuario.
- La información es cacheada al mismo tiempo que es distribuida a los usuarios sin tener que obtenerla desde el servidor en cada requerimiento.
- Mayor tiempo de respuesta de las aplicaciones

Las arquitecturas orientadas a servicios (SOA - Service Oriented Architecture) permiten el desarrollo de aplicaciones que consisten de un complejo conjunto de servicios discretos que dan servicio a uno o mas procesos de negocios. Aunque SOA ha conllevado a una cambio radical en la filosofía del software, distribuir servicios a los usuario en una manera perceptiva ha probado ser un gran desafío. Una manera simple para que una aplicación RIA utilice un servicio Web es mediante la tecnología AJAX.

3.5. Tecnologías para el desarrollo RIA

Se han propuesto varias tecnologías para soportar el desarrollo RIA. Aunque son diferentes en muchos aspectos, se pueden clasificar en 4 grandes categorías:

1. `Basadas en scripts (Scripting-based)`: la lógica del lado del cliente es implementada por medio de lenguajes de *scripts*, tal como *JavaScript*, y las interfaces están basadas en una combinación de HTML y hojas de estilo CSS.

2. `Basadas en plugins (Plugin-based)`: la renderización avanzada y el procesamiento de eventos es soportada por *plugins* en los navegadores que interpretan XML, programas de propósito general o archivos multimedia.

3. `Basadas en navegadores (Browser-based)`: donde la interacción enriquecida es soportada en forma nativa por algunos navegadores que interpretan lenguajes de definición de interfaces declarativos (XUL).

4. `Escritorios basados en Web (Web-based desktop)`: las aplicaciones son descargas desde la Web pero ejecutadas fuera del contexto de un navegador Web.

A continuación describiremos brevemente algunas de estas tecnologías, y especificando a que categoría pertenecen.

3.5.1. Ajax

AJAX es el acrónimo de *Asynchronous JavaScript y XML*. AJAX no es una tecnología en sí mismo sino que se trata de la agrupación de varias tecnologías:

- `XHTML y CSS`: para crear una presentación basada en estándares.
- `DOM`: para la interacción y manipulación dinámica de la presentación.
- `XML, XSLT y JSON`: para el intercambio y la manipulación de información.
- `Objeto XMLHttpRequest`: para el intercambio asíncrono de información.
- `JavaScript`: que permite combinar todas las tecnologías anteriores.

AJAX pertenece a la categoría de tecnologías RIA basadas en *scripts*. En la figura 3.1 se observa del lado izquierdo el modelo tradicional de las aplicaciones Web, mientras que del lado derecho se observa el modelo propuesto por AJAX.

AJAX, al igual que la mayoría de las tecnologías para el desarrollo RIA, permite mejorar completamente la interacción del usuario con la aplicación, evitando las recargas constantes de las páginas, ya que el intercambio de información con el servidor se produce en segundo plano.

Las aplicaciones construidas con AJAX eliminan la recarga constante de páginas mediante la creación de un elemento intermedio entre el usuario y el servidor. La capa intermedia de AJAX mejora la respuesta de la aplicación, ya que el usuario nunca se encuentra con una ventana del navegador vacía esperando la respuesta del servidor.

La figura 3.2 muestra la diferencia más importante entre una aplicación Web tradicional y una aplicación Web creada con AJAX. La imagen superior muestra la interacción síncrona propia de las aplicaciones Web tradicionales. La imagen inferior muestra la comunicación asíncronica de las aplicaciones creadas con AJAX.

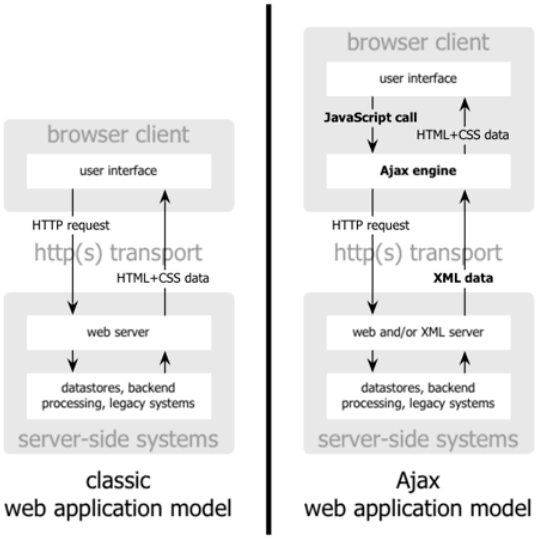

Figura 3.1: Comparación del modelo tradicional de aplicación Web y el modelo propuesto por AJAX

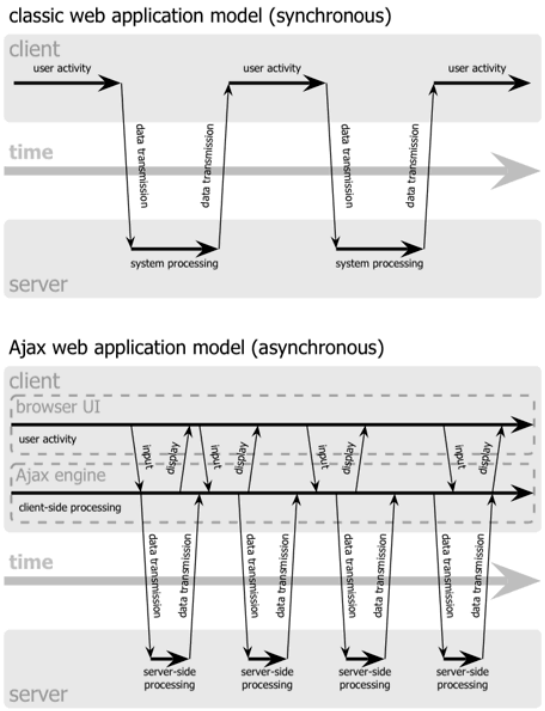

Figura 3.2: Comparación entre las comunicación síncronica de las aplicaciones Web tradicionales y la comunicación asíncronica de las aplicaciones AJAX

Las peticiones HTTP al servidor se sustituyen por peticiones *JavaScript* que se realizan al elemento encargado de AJAX. Las peticiones más simples no requieren intervención del servidor, por lo que la respuesta es inmediata. Si la interacción requiere una respuesta del servidor, la petición se realiza de forma asíncronica. En este caso, la interacción del usuario tampoco se ve interrumpida por recargas de página o largas esperas por la respuesta del servidor.

Los conceptos sobre comunicación asincrónica descriptos en esta sección son propios de las aplicaciones RIA en general y se aplican a las otras tecnologías que describiremos en las siguientes secciones.

Desde su aparición, se han creado cientos de aplicaciones Web basadas en AJAX. En la mayoría de los casos, AJAX puede sustituir completamente a otras tecnologías como *Flash*. Además, en el caso de las aplicaciones Web más avanzadas, pueden llegar a sustituir a las aplicaciones de escritorio.

Como ejemplo podemos citar al correo electrónico Web *Gmail* [Gmail09] de *Google*, el cual esta completamente desarrollado con la tecnología AJAX (figura 3.3).

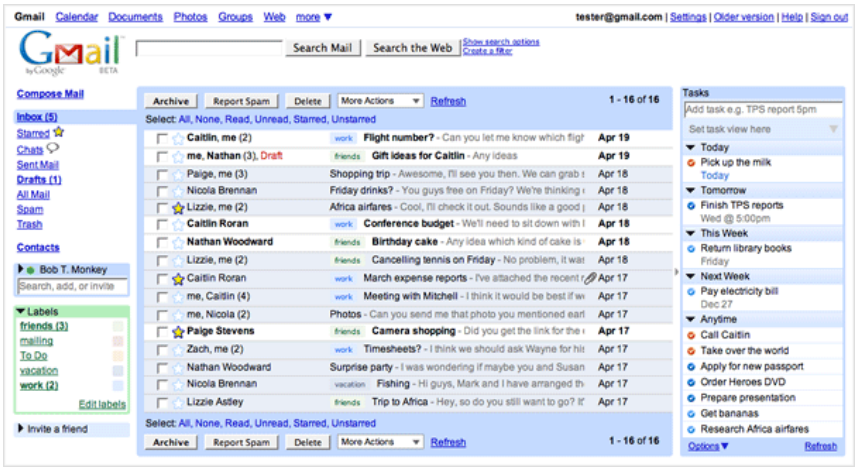

Figura 3.3: `Gmail: el correo electrónico Web de Google, desarrollado con la tecnología AJAX`

Yahoo! User Interface (YUI) library

La librería YUI [YUI09] es un conjunto de utilidades y controles, escritos en JavaScript, para la construcción de aplicaciones RIA mediante la utilización de técnicas como DOM Scripting, DHTML y AJAX. YUI esta disponible bajo la licencia BSD, con lo cual puede ser descargada gratuitamente.
La librería fue lanzada en 2005 y fue convertida en open source bajo la licencia BSD en Febrero de 2006. Desde su lanzamiento, la librería fue integrada en la mayoría de los sitios propietarios de Yahoo!, incluyendo su página principal, Yahoo! Mail [YMail09] y Flickr [YFlickr09].

Yahoo! Design Pattern library

Por su parte, Yahoo! Design Pattern Library [YDPL09] es una colección de patrones en interfaces de usuario, mas específicamente en interfaces de aplicaciones RIA. Este catalogo agrupa a los patrones de diseño en diferentes categorías:

- `Navegación` (Pagination, Tabs)
- `Distribución de la información` (Page Grids)
- `Selección` (Autocomplete, Carousel)
- `Interacción` (Hover tool tip, Hover detail)
- `Sociales` (Rating an object, Numbered labels)

En los patrones no se especifica como implementarlos con alguna tecnología en particular, los mismos pueden ser implementados con la mayoría de las tecnologías disponibles para aplicaciones RIA. En particular, la librería YUI [YUI09] posee componentes que implementan cada uno de los patrones de diseño.
En la segunda parte de este trabajo, haremos uso de estos patrones de diseño para definiendo tests de navegación que verifiquen su correcto funcionamiento dentro de una aplicación RIA.

3.5.2. Adobe Flash

Adobe Flash (previamente conocido como Macromedia Flash) es una plataforma multimedia originalmente adquirida por Macromedia y actualmente desarrollada y distribuida por Adobe Systems [Adobe09]. Desde sus comienzos en 1996, Flash se ha convertido en una tecnología muy popular para agregar animaciones e interactividad a las páginas Web. Flash es comúnmente utilizado para crear animaciones, avisos e integrar vídeo a las páginas Web, y mas recientemente para desarrollar aplicaciones RIA. Según nuestra clasificación de tecnologías RIA, Adobe Flash pertenece a las tecnologías basadas en plugins.

Flash puede manipular gráficos vectorizados y rasterizados, y soporta *streaming* bidireccional de audio y vídeo. Contiene un lenguaje de *scripting* llamado *ActionScript*. La variedad de productos de software, sistemas y dispositivos que son capaces de crear o mostrar contenido Flash es muy amplia, incluyendo el reproductor Adobe Flash que esta disponible gratuitamente para la mayoría de los navegadores, algunos dispositivos móviles y otros dispositivos electrónicos (mediante el uso de Flash Lite). El software de autoría de Adobe Flash Profesional es utilizado para crear contenido para la plataforma Adobe Engagement, tal como aplicaciones Web, juegos y películas, y contenido

para teléfonos móviles y otros dispositivos embebidos.

Los archivos con formato `swf`, tradicionalmente llamados películas *Flash ShockWave*, películas Flash o juegos Flash, usualmente tienen la extensión *.swf* y pueden ser un objeto embebido en una pagina Web, estrictamente reproducidos en un reproductor Flash, o incorporados en un proyector (película Flash auto ejecutable, con la extensión *.exe* en Microsoft Windows o *hqx* en Machintosh). Los archivos de vídeo tienen la extensión *.flv* y pueden ser reproducidos desde un archivo *swf* o algún reproductor como VLC.

Actualmente son muchos los sitios Web que están desarrollados completamente en Flash, mientras que otros sitios desarrollados en otras tecnologías incluyen algún tipo de contenido (por ejemplo, *banners* publicitarios) desarrollado con Flash. Como ejemplo de un sitio Web desarrollado íntegramente con Flash, podemos nombrar el sitio *Speak Visual* [NVIDIA09] de NVIDIA (figura 3.4).

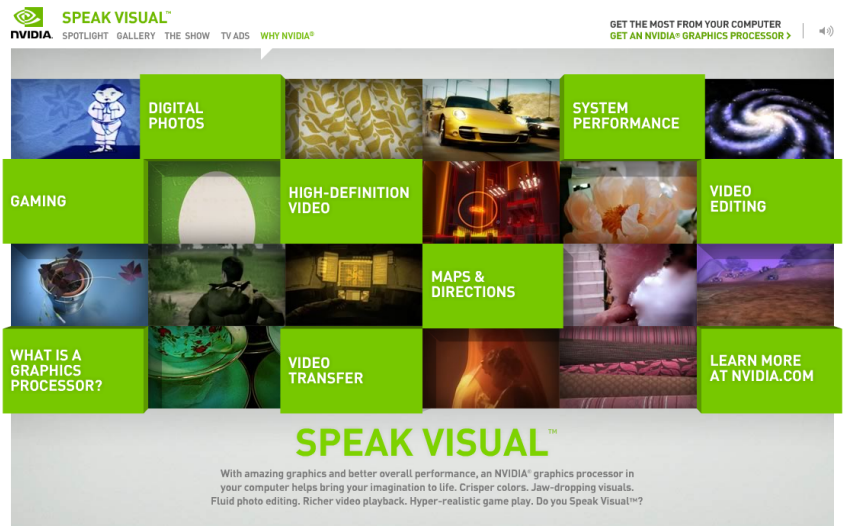

Figura 3.4: `Sitio Web desarrollado en Adobe Flash: NVIDIA` *`Speak Visual`*

3.5.3. Adobe Flex

Adobe Flex es un kit de desarrollo de software, distribuido por Adobe Systems [Adobe09], para el desarrollo y distribución de aplicaciones RIA multi-plataforma. Las aplicaciones Flex pueden ser escritas utilizando Adobe Flex Builder o algunos de los compiladores de distribución gratuita, también provistos por Adobe. Al igual que Adobe Flash, es una tecnología para el desarrollo RIA basada en plugins.

El lanzamiento inicial, de Marzo de 2004 por parte de Macromedia, incluía un kit de desarrollo de software, un IDE y una aplicación de integración con J2EE conocida como Flex Data Services. Desde que Adobe adquirió Macromedia en 2005, las nuevas distribuciones de Flex ya no necesitaban una licencia para el componente Flex Data Services, ya que el mismo se convirtió en un producto separado y renombrado a LiveCycle Data Services.

En febrero de 2008, Adobe distribuye el SDK Flex 3 bajo la licencia open source *Mozilla Public License*. Sin embargo, el reproductor Adobe Flash (en el cual son visualizadas las aplicaciones Flex) y el Adobe Flex Builder (el IDE para construir aplicaciones Flex) continúan siendo propietarios de Adobe.

Como ejemplo de un sitio Web desarrollado en Adobe Flex, se puede nombrar al sitio de

productos deportivos *Wilson* [Wilson09] (figura 3.5). Este sitio esta completamente desarrollado con esta tecnología, en la figura 3.5, se puede observar el catalogo de raquetas de tenis donde al seleccionar un modelo en particular se muestra en el lado izquierdo las características e imagen ampliada de la misma; todo esto sucede sin re actualizar toda la pagina en cada selección de un nuevo modelo de raqueta.

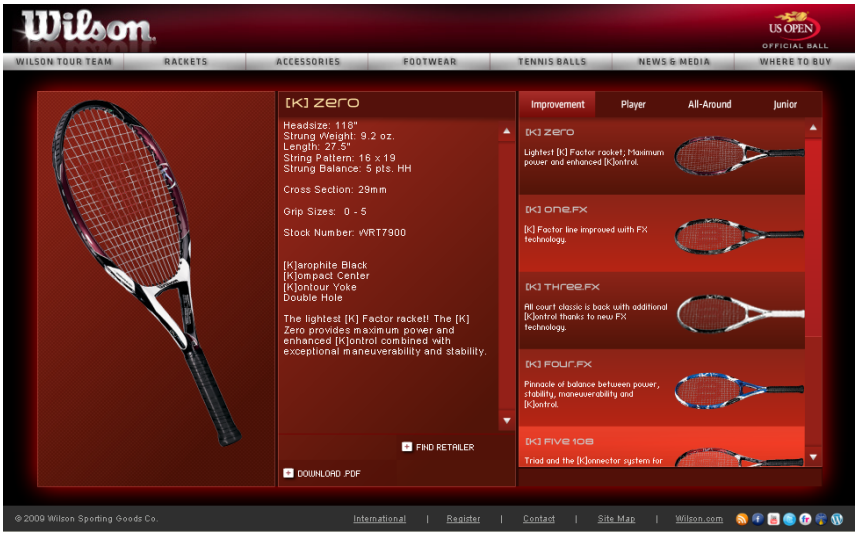

Figura 3.5: `Sitio Web desarrollado en Adobe Flex: Wilson`

3.5.4. Adobe AIR

Adobe Integrated Runtime (AIR) es entorno de ejecución multi-plataforma para la construcción de aplicaciones RIA mediante la utilización de Flash, Flex, HTML o AJAX; que pueden ser distribuidas como aplicaciones de escritorio tradicionales. Adobe AIR pertenece al grupo de tecnologías RIA de escritorios basados en Web.

Adobe realizo el primer lanzamiento de AIR (llamado previamente Apollo), junto con un kit de desarrollo (SDK) y una extensión para desarrollar aplicaciones AIR con el *framework Flex*, en Marzo de 2007. Para junio de 2007, el producto se renombró a AIR y se lanzo la distribución beta. Para diciembre de 2008 se distribuyó la primer versión estable para Linux.

AIR tiene como objetivo ser un entorno de ejecución versátil, que permite al código Flash, Action Script o HTML y Javascript ser utilizado para la construcción de aplicaciones de escritorio tradicionales. La idea es tener un entorno de ejecución para aplicaciones RIA que puedan ser ejecutadas sobre el escritorio sin la necesitad de un navegador Web. La diferencia con una aplicación RIA utilizada mediante un navegador es que esta ultima no requiere ningún tipo de instalación, mientras que una aplicación distribuida con AIR necesita ser empaquetada, firmada digitalmente e instalada en el sistema de archivos local. Esto permite a las aplicaciones tener acceso al almacenamiento local y sistema de archivos, que suele ser bastante limitado para una aplicación ejecutada desde un navegador.

Las aplicaciones AIR pueden operar fuera de línea, y luego activar funcionalidades extra cuando este disponible una conexión a Internet.

Como ejemplo de una aplicación Adobe AIR se puede citar a la aplicación *AOL Music Top 100* [AOLMT09] (figura 3.6), la cual permite visualizar y escuchar vídeos musicales.

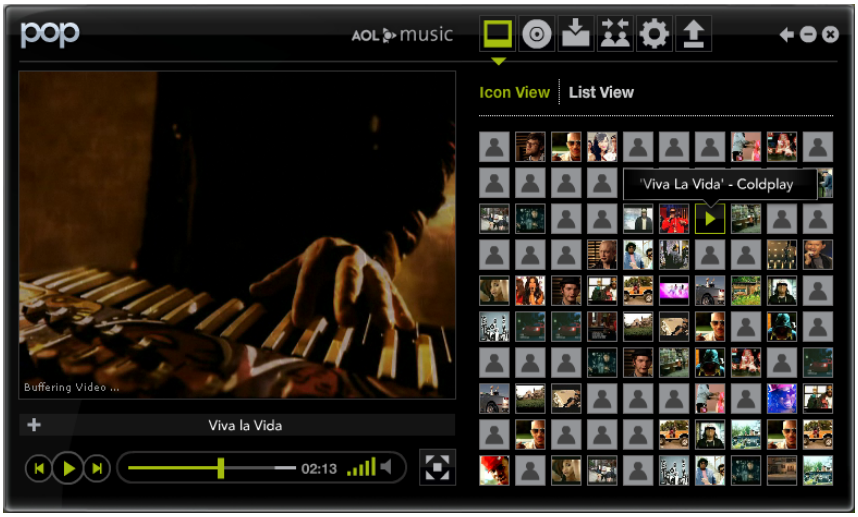

Figura 3.6: `Aplicación desarrollada en Adobe AIR: AOL Music Top 100 Videos`

3.5.5. JavaFX

JavaFX [JavaFX09] es una plataforma que permite el desarrollo y distribución de aplicaciones RIA. JavaFX fue lanzado en Diciembre de 2008 y esta completamente integrado con el entorno de ejecución Java. Las aplicaciones desarrolladas con JavaFX pueden ser ejecutadas tanto en el escritorio como un navegador con soporte Java. Por este motivo esta tecnología RIA cae dentro de dos categorías RIA: basadas en plugins y escritorios basados en Web. Las aplicaciones son escritas utilizando un lenguaje de *scripts* estáticamente tipeado y declarativo llamado JavaFX Script que facilita la programación en un contexto visual permitiendo a los desarrolladores crear GUIs avanzadas rápida y fácilmente. La plataforma provee un amplio conjunto de librerías para la manipulación de gráficos, multimedia y Web services. Ademas, es posible utilizar cualquier librería Java tradicional.

La plataforma JavaFX 1.2 incluye los siguientes componentes:

- `JavaFX SDK`: que incluye el compilador y las herramientas, gráficos, multimedia, Web services, controles de interfaz y librerías de texto para crear aplicaciones RIA, tanto para navegadores, escritorio y plataformas móviles.
- `NetBeans IDE 6.5.1 para JavaFX 1.2`: provee un entorno de desarrollo integrado para la construcción, previsualización y debugging de las aplicaciones. El editor posee un paleta *drag and drop* que permite agregar objetos con transformaciones, efectos y animaciones de manera fácil y rápida. El editor también provee un conjunto amplio de ejemplos.
- `JavaFX 1.2 Production Suite`: es una suite de herramientas y plugins que permite exportar elementos gráficos a las aplicaciones JavaFX. La suite consiste de:
  - `JavaFX 1.2 plugins for creative tools`: son un conjunto de plugins para *Adobe Photoshop CS3* y *Adobe Illustrator CS3* que permiten exportar elementos gráficos desde estas aplicaciones a código JavaFX Script.
  - `JavaFX 1.2 Media Factory`: son un conjunto de herramientas para convertir gráficos SVG a código JavaFX Script y previsualizar elementos gráficos importados a JavaFX desde otras herramientas.

JavaFX es una tecnología relativamente nueva y su intención es competir con otras tecnologías ya instaladas en el mercado como son Flash de Adobe, y Silverlight de Microsoft. Se pueden observar varias aplicaciones de ejemplo desarrolladas con esta tecnología en el sitio de JavaFX [JavaFX09].

3.5.6. Microsoft Silverlight

Microsoft Silverlight [Silv09] es un framework para el desarrollo de aplicaciones Web de la empresa Microsoft. Inicialmente fue lanzado como un *plugin* para *streaming* de vídeo; las sucesivas versiones agregaron nuevas características de interactividad y soporte para .NET y herramientas de desarrollo.

Silverlight es compatible con múltiples navegadores Web utilizados en los sistemas operativos Microsoft Windows y Mac OSX. Existe una implementación gratuita llamada *Moonlight*, desarrollada por Novel en cooperación con Microsoft, para permitir la compatibilidad con el sistema operativo Linux.

Silverlight provee un sistema de gráficos similar al de *Windows Presentation Foundation*, e integra multimedia, gráficos, animaciones e interactividad en un solo complemento. En las aplicaciones Silverlight, las interfaces de usuario son definidas en XAML y programadas utilizando un subconjunto del framework .NET. El contenido textual creado con Silverlight puede ser indexado y luego ser incluido en las indices de los motores de búsqueda ya que no es compilado, pues se representa con XAML. Silverlight también puede ser utilizado para crear *Windows Sidebar gadgets* para el sistema operativo Windows Vista. Silverlight pertenece a las categorías de tecnologías RIA basadas en navegadores y escritorios basados en Web.

Silverlight permite cargar dinámicamente contenido XML que puede ser manipulado, a través de una interfaz DOM, de una manera consistente con las técnicas AJAX convencionales. Silverlight tiene un *Downloader* para solicitar *scripts* u otros medios y guardarlos en la maquina cliente, cuando los mismos son requeridos por la aplicación.

La figura 3.7 muestra una aplicación desarrollada con Microsoft Silverlight, se trata de un buscador de autos del sitio *kbb* [Kbb09]. En esta aplicación, es posible buscar autos de determinadas características. Luego de iniciar la búsqueda se pueden observar en forma de mosaico todas las fotos de los autos que cumplen con los parámetros de búsqueda. Es posible hacer un *zoom* de cada imagen y ver la información de cada vehículo en particular.

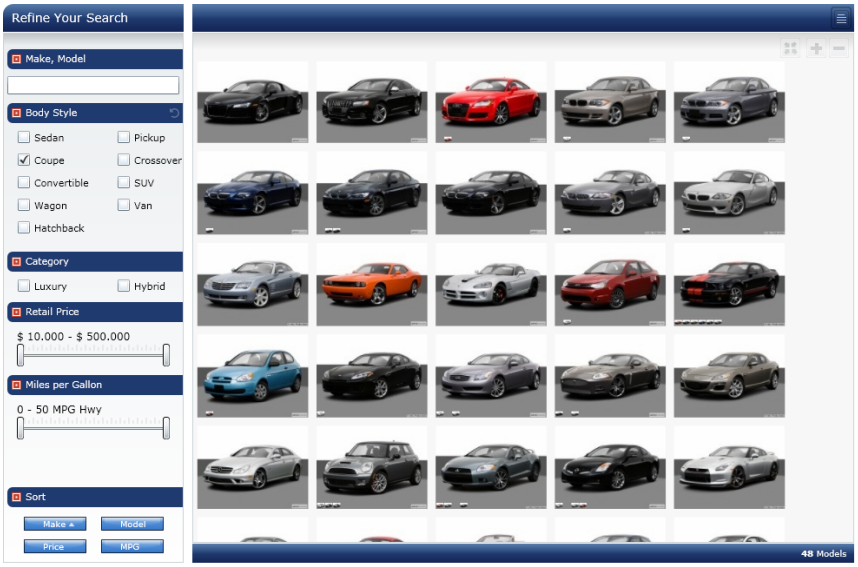

Figura 3.7: `Aplicación desarrollada en Microsoft Silverlight: KBB PerfectCarFinder`

3.5.7. OpenLaszlo

OpenLaszlo [Laszlo09] es una plataforma para el desarrollo y distribución de aplicaciones RIA. Es distribuido por Laszlo Systems bajo la licencia *open source Common Public License* (CPL).

La plataforma OpenLaszlo consiste de un lenguaje de programación llamado LZX y un servidor (OpenLaszlo Server):

- LZX es un lenguaje descriptivo de XML y Javascript similar a XUL, MXML, y XAML. LZX permite llevar a cabo un proceso de desarrollo basado en texto que soporta la prototipación rápida y las buenas practicas de desarrollo de software. Esta diseñado con el objetivo de resultar familiar a los desarrolladores de aplicaciones Web tradicionales, que están familiarizados con HTML y JavaScript.
- OpenLaszlo Server es un servlet Java que compila aplicaciones LZX es binarios ejecutables.

OpenLaszlo se destaca, de otras tecnologías disponibles en el mercado, por ser el único que permite derivar de un mismo código aplicaciones basadas en AJAX y Flash. Por este motivo, no podemos clasificar esta tecnología en alguna de las categorías antes mencionadas, ya que podemos ver a OpenLaszlo como un meta-lenguaje de aplicaciones RIA. Dependiendo de la derivación a la tecnología RIA en concreto que realicemos tendremos la clasificación particular.

Entre los sitios que utilizan esta tecnología se puede destacar a la cadena de tiendas *Walmart* [Walm09] (figura 3.8).

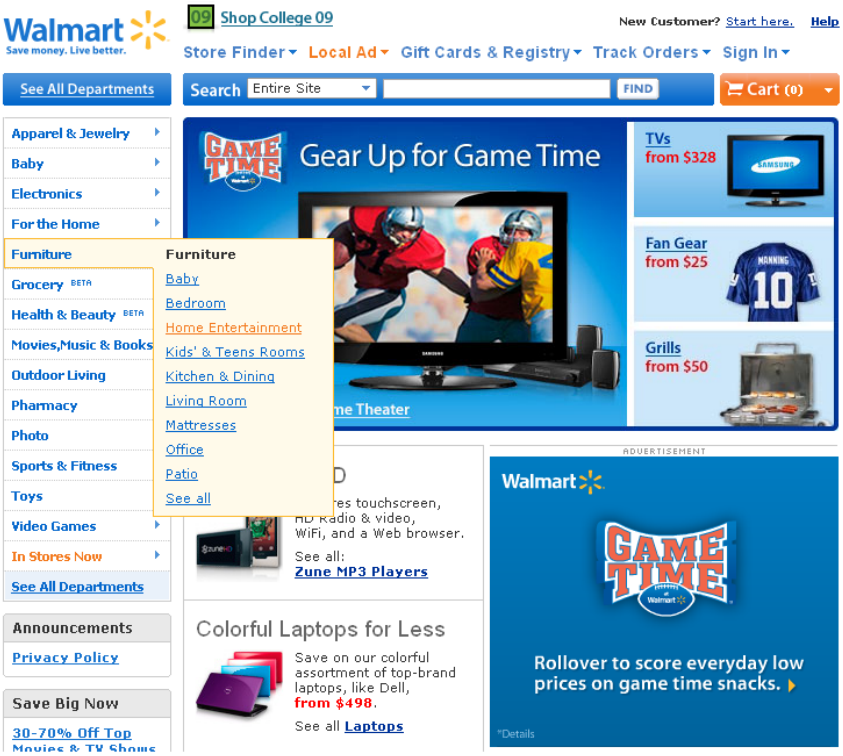

Figura 3.8: `Sitio desarrollado con Open Laszlo: Walmart`

3.6. Resumen

Las aplicaciones RIA han adquirido gran popularidad en los últimos años. Los factores que mas influyeron en este crecimiento son el aumento del ancho de banda, el aumento de la capacidad de computo de las máquinas clientes, el interés por esta tecnología por parte de grandes companías como Google y el desarrollo de las tecnologías de Web Services y las arquitecturas orientadas a servicios (SOA).

Las aplicaciones RIA reúnen las características distintivas de las aplicaciones tradicionales de escritorio y las aplicaciones Web. Su potencial se basa en su gran accesibilidad, continuidad visual, reducción del trafico de red, interfaces de usuario mas completas e interactivas, respuestas en tiempo real y usabilidad.

Existen numerosos frameworks, algunos bajo licencias pagas y otro open source, para el desarrollo de aplicaciones RIA. Cada uno presenta diferentes características y funcionalidades a los desarrolladores. Se pueden clasificar estas tecnologías en 4 grande grupos: 1) Basadas en scripts, 2) Basadas en plugins, 3) Basadas en navegadores, y 4) Escritorios basados en Web.

Entre los frameworks mas utilizados y populares de la actualidad se puede nombrar a Adobe Flash, Adobe Flex, Adobe AIR, Ajax, JavaFX, Microsoft Silverlight y OpenLaszlo.

Capítulo 4

User Interaction Diagrams

Los UIDs (User Interaction Diagrams) [VSdS00] son una técnica diagramática para representar el intercambio de información durante la interacción entre el usuario y una aplicación. Los UIDs han probado ser una técnica de gran valor para la captura de requerimientos, ya que describen el intercambio de información en una alto nivel de abstracción.

Los UIDs pueden ser incorporados al proceso de diseño de una aplicación Web. Durante la captura de requerimientos, pueden ser utilizados para representar los requerimientos; en la etapa de diseño conceptual, pueden servir como la base para derivar un diagrama de clases preliminar utilizando lineamientos heurísticos. Ademas, en el diseño navigacional, se pueden utilizar lineamientos heurísticos adicionales para especificar esquemas de contexto parciales a partir de estos diagramas.

4.1. Introducción

Una de las características distintivas de las aplicaciones Web con respecto a las aplicaciones de sistemas de información tradicionales es la navegación. Mientras que ambas organizan la información en elementos con alguna relación o significado las aplicaciones Web permiten al usuario navegar a través de los elementos (hipervínculos) utilizando su estructura navigacional.

UML (*Unified Modeling Language*) sugiere utilizar los casos de uso para modelar la interacción con el usuario durante la captura de requerimientos. Las técnicas textuales como los escenarios y casos de uso, pueden producir especificaciones completas y detalladas, y son usualmente fáciles de entender por personas sin conocimientos técnicos. Por otra parte, en el caso de grandes aplicaciones, tales especificaciones suelen ser extensas, ambiguas y sobrecargadas de información; lo cual las hace menos entendibles y mas dificultosas de administrar.

Las técnicas diagramáticas existentes, tales como los diagramas de secuencia, colaboración, estados, actividad, flujos de datos y aun los esenciales casos de uso no son capaces de enfocarse en la interacción y al mismo tiempo satisfacer todos estos requerimientos. UML especifica la realización de casos de uso utilizando diagramas de secuencia, colaboración y estado. Sin embargo, estos diagramas son mas apropiados para el diseño de sistemas (e implementación) que para la especificación de requerimientos, ya que innecesariamente fuerzan a decisiones de diseño tempranas, tales como la identificación de objetos.

Los diagramas UIDs pueden ser utilizados para minimizar esta brecha, enfocándose exclusivamente en el intercambio de información entre la aplicación y el usuario. También, como se menciono anteriormente, los UIDs pueden ser utilizados en el proceso de diseño de aplicaciones Web en niveles incrementales de detalle, comenzado con la captura y especificación de requerimientos. En esta

etapa inicial, la técnica utilizada debería presentar la interacción de la información (entre el usuario y la aplicación) en una manera estructurada, detallada y diferenciada; permitir una comunicación fluida entre los diseñadores y sus clientes y entre los diseñadores y los programadores; y soportar trazabilidad desde los requerimientos hasta los posteriores pasos de diseño. La figura 4.1 muestra la utilización de los UIDs a través del proceso de diseño de una aplicación Web.

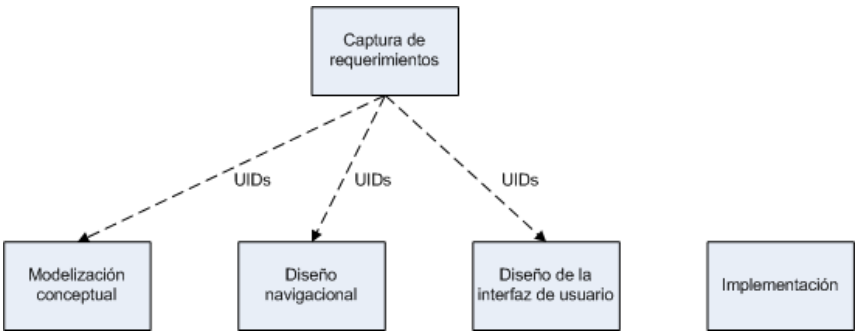

Figura 4.1: `Los UIDs en el proceso de diseño de una aplicación`

4.2. Estructura

Los diagramas UIDs pueden ser vistos como una descripción diagramática del intercambio de información que es textualmente descripta en un caso de uso. La notación de los UIDs es bastante simple, ya que uno de sus objetivos es que puedan ser fácilmente interpretados por los usuarios y demás interesados.

Los UIDs se componen de un conjunto de estados de interacción conectados por transiciones. Los estados de interacción representan los intercambios de información entre el usuario y la aplicación, mientras que las transiciones representan el cambio de foco o contexto de una interacción a otra. Se puede decir que un estado de interacción se convierte en un *foco de interacción* cuando la información dentro del mismo representa la información que esta siendo actualmente intercambiada entre el usuario y la aplicación. Las transiciones suceden, en general por información ingresada o seleccionada por el usuario. A continuación mostraremos algunos ejemplos de diagramas UIDs. Los ejemplos de este capítulo y los posteriores están basados en un sitio de ventas de CDs musicales, similar a la sección de ventas de CDs del sitio de *e-commerce Amazon* [Amazon09]. En el apéndice A se presenta la notación completa de los diagramas UIDs.

La figura 4.2 muestra el UID (creado con el editor RIA-UID) definido para representar las interacciones en la tarea de *Seleccionar un CD en base al genero musical*. En esta tarea, el usuario selecciona un tipo (genero) de música y el sistema retorna un conjunto de CDs pertenecientes al genero musical seleccionado. De este conjunto, el usuario puede agregar al carrito de compras los CDs que posteriormente quiera comprar.

En el estado de interacción inicial, el sistema presenta al usuario un conjunto de géneros musicales. Durante este estado de interacción, el usuario debe seleccionar un tipo de música seguido de una de las opciones: *artistas* (artists), *todos los CDs* (all CDs) o *sugerencias* (suggestions). Si el usuario selecciona la opción *artistas*, entonces el estado de interacción que muestra el conjunto de artistas (para el genero de musical seleccionado) se convierte en el foco de la interacción. Siguiendo este estado de interacción, el usuario debe seleccionar uno o mas artistas y el sistema mostrará un conjunto de CDs relacionados a dichos artistas. De la misma manera, el conjunto de CDs relacionados al artista seleccionado es presentado en un estado de interacción separado ya que

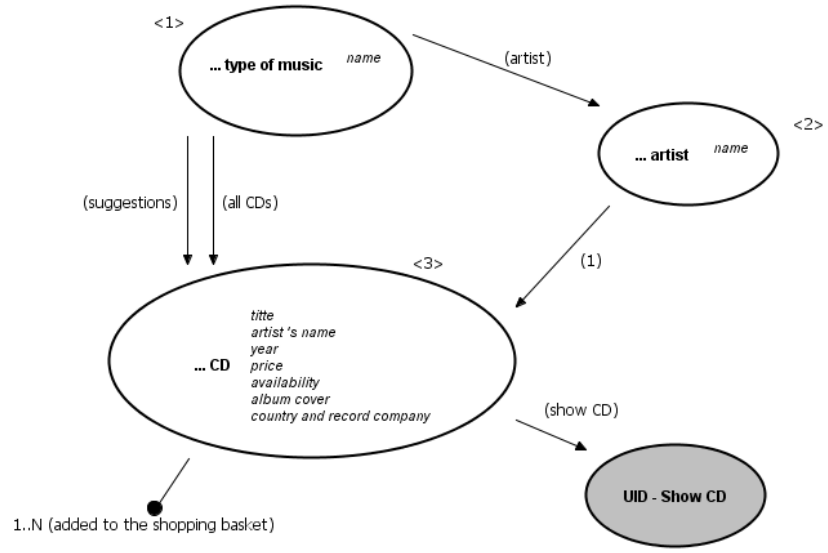

Figura 4.2: `UID - Selección de un CD en base al genero musical`

depende de la selección realizada previamente.

Si el usuario selecciona la opción *sugerencias* (suggestions), entonces el sistema muestra el conjunto de CDs recomendados para el genero musical seleccionado. Si el usuario selecciona la opción *todos los CDs* (all CDs), entonces el sistema mostrará todos los CDs pertenecientes al genero musical seleccionado.

El conjunto de CDs se representa por medio de una estructura, que contiene los ítems de información: *titulo, nombre del artista, año, precio, disponibilidad, imagen del álbum, país, y compañía discográfica*. Como es posible que halla mas de un CD para los parámetros seleccionados (genero y opción), esto se especifica mediante la cardinalidad, en este caso `...` indica que puede haber varios CDs (también es posible indicarlo de la forma `1..N`). Desde el estado de interacción `<3>`, es posible seleccionar 1..N CDs para agregarlos al carrito de compras, o seleccionar un CD para ver su información detallada. En este último caso se transfiere el foco de interacción desde un UID a otro UID, para el caso de nuestro ejemplo se transfiere el foco al diagrama UID `Show CD`.

4.3. Captura de requerimientos

Los UIDs son definidos a partir de los escenarios o casos de uso. Se genera un diagrama por cada escenario o caso de uso definido. Para poder obtener un diagrama, a partir de un escenario o caso de uso, se deben identificar y organizar en interacciones los intercambios de información entre el usuario y el sistema. La identificación de esta información es de suma importancia ya que es la base para la definición del diagrama. La secuencia de pasos que describiremos a continuación pueden servir de guía para traducir un escenario o caso de uso en un diagrama UID.

1. Se comienza analizando el escenario o caso de uso para identificar los ítems de información intercambiados, entre el usuario y el sistema, que generalmente suelen ser los sustantivos. Es importante también, identificar que ítems de información son ingresados por el usuario y cuales retornados por el sistema. Para ilustrar la idea tomaremos como ejemplo el caso de uso de *seleccionar un CD en base al nombre del artista*. En **negrita** se resaltan los ítems de información ingresados por el usuario y en cursiva los ítems de información retornados por

el sistema.

> `Caso de uso - Selección de un CD en base al nombre del artista`
>
> El usuario ingresa el **nombre del artista** y opcionalmente puede ingresar **el año de edición del CD**. El sistema retorna una lista con los nombres de los artistas, cuyo nombre coincide con el ingresado por el usuario; si solo hay una coincidencia, directamente se muestran *los CDs para dicho artista*. El usuario selecciona el **artista de su interés** y el sistema retorna la lista de *CDs disponibles*. Para cada CD, se muestra *su titulo, nombre de el/los artista/s, año, precio, disponibilidad, y una imagen del albúm*. El usuario puede acceder a mas información sobre el CD (ver caso de uso: `Show CD`). También esta disponible la opción de comprar uno o mas CDs, para esto el usuario debe seleccionar los CDs y agregarlos al carrito de compras para realizar posteriormente la transacción (caso de uso: `Comprar CDs`)

2. Luego de identificar la información intercambiada, se dividen los ítems de información en estados de interacción. Los ítems de información son ubicados en el mismo estado de interacción, a menos que dependan de los ítems previos o se deba seleccionar una opción; en tales casos, deben ser ubicados en un estado de interacción diferente. Sin embargo, la información retornada por el sistema que precede a información ingresada por el usuario no puede ser ubicada en el mismo estado de interacción. Luego, se intenta ubicar los ítems restantes en el siguiente estado de interacción, y se procede de la misma manera hasta poder ubicar todos los ítems. Los escenarios y casos de uso referenciados, desde el caso de uso o escenario que esta siendo analizado, son representados por llamadas a otros UIDs. En el ejemplo, comenzamos con el ítem de datos *nombre del artista*, que es ubicado en el estado de interacción ¡1¿del UID. El siguiente ítem de datos intercambiado es el *año del CD*, el cual no depende del ítem previo (nombre del artista), por lo tanto se ubica en el mismo estado de interacción. Ya que el siguiente ítem de datos, el conjunto de *nombres de compositores*, depende del nombre del artista y año ingresado, es ubicado en el siguiente estado de interacción ¡2¿, y así se continua con los ítems restantes.

3. Los ítems de datos ingresados por el usuario y los retornados por el sistema deben ser diferenciados. De esta manera, los ítems de datos obligatorios ingresados por el usuario son representados dentro de rectángulos con un borde continuo, mientras que los ítems opcionales son representados con un rectángulo con borde punteado. Los ítems retornados por el sistema son ubicados directamente en la interacción, sin ningún rectángulo alrededor de los mismos. Para las estructuras se listan todos lo ítems que la conforman y previo al nombre se puede especificar su cardinalidad.

4. Los estados de interacción son conectados por medio de transiciones. Es posible conectar un estado de interacción con dos o mas estados de interacción diferentes, permitiendo representar varias alternativas. En este caso, la información ingresada por el usuario determina que alternativa de interacción será la siguiente en obtener el foco de interacción. Si el cambio del foco de interacción es el resultado de la selección un elemento, el número de elementos seleccionados puede ser agregado a la flecha, y la fuente de esta flecha es dirigida desde donde provienen los elementos seleccionados.

5. Las operaciones ejecutadas por los usuarios son identificadas y representadas por opciones. Son generalmente, representadas en los escenarios y casos de uso por verbos. Las opciones son representadas en el diagrama por etiquetas adjuntadas a las transiciones. Si luego de la

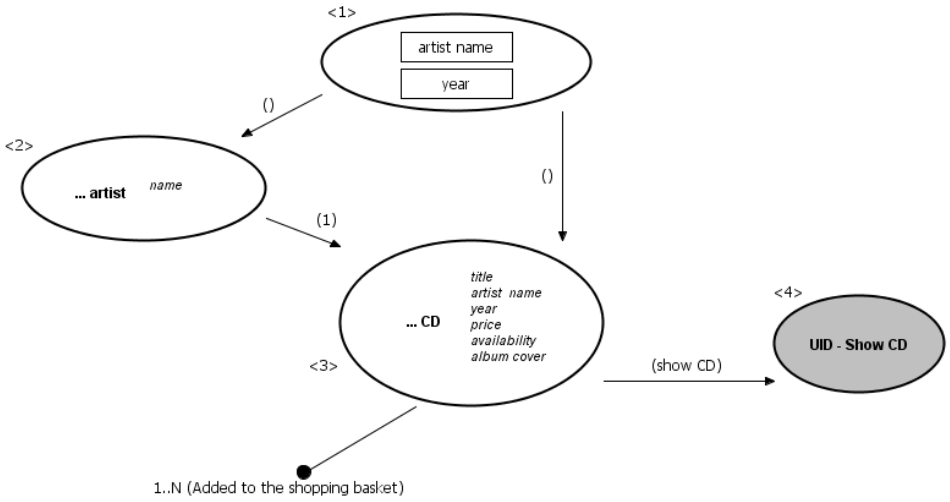

Figura 4.3: `UID - Selección de CDs en base al nombre del compositor`

selección de una opción, el foco de la interacción no cambia, entonces las opciones pueden ser representadas por una línea con un círculo.

6. Los requerimientos no funcionales no son representados gráficamente en los UIDs. Aunque los caso de uso no especifican estos requerimientos, en ocasiones pueden ser nombrados. En este caso, podrían ser adjuntados al UID con una nota textual. La figura 4.3, es la representación completa del caso de uso definido en esta sección.

Luego de definir los UIDs, el diseñador interactúa con cada usuario para validar los casos de uso y sus correspondientes diagramas UIDs. Para lograr este objetivo se debe mostrar los casos de uso y los UIDs para verificar que el usuario esta de acuerdo con los mismos; esto permite que no sea necesario tratar con requerimientos de usuario conflictivos. Es importante explicar la notación de los UIDs a los usuarios. En general, los usuarios pueden aprender fácilmente la notación y estar en condiciones de interactuar con el diseñador para expresar su entendimiento de la tarea que esta siendo modelada a través del diagrama.

4.4. Resumen

Los UIDs son una técnica diagramática que permite representar los intercambios de información que existen entre un usuario y una aplicación. Estos diagramas complementan a otras técnicas para la captura de requerimiento como los escenarios, casos de uso y diagramas UML. Su principal característica es la de ser una notación muy simple y fácil de entender por parte de los usuarios, lo cual permite una comunicación fluida entre el diseñador y los usuarios para lograr especificar los requerimientos lo mas adecuadamente posible. Se puede aplicar una serie de pasos definidos para derivar un diagrama UID a partir de un caso de uso o escenario.

Los UIDs son una herramienta útil para el desarrollo de aplicaciones Web. Pueden utilizarse en varias etapas del desarrollo incluyendo el modelado conceptual, el diseño navigacional y el diseño de la interfaz de usuario.

Capítulo 5

MDWE y TDD en el desarrollo de aplicaciones Web

En este capitulo describiremos una metodología (propuesta inicialmente en [RRG09a]) que permite combinar el estilo ágil, iterativo e incremental de TDD con las metodologías de ingeniería Web basadas en transformaciones y dirigidas por modelos MDWE; enfocándose no solo en el proceso de desarrollo sino también en la evolución de la aplicación, permitiendo que los test puedan ser transformados y adaptados en conjunto con el refactoring de los modelos.

5.1. Metodologías ágiles e ingeniería Web

Las metodologías ágiles son particularmente atractivas para el desarrollo de aplicaciones Web. En estas metodologías las aplicaciones son construidas en forma incremental, con una alta participación de los diferentes interesados (desarrolladores, clientes y usuarios) en la validación de los sucesivos prototipos generados. Por su parte, los enfoques de ingeniería Web dirigidos por modelos (*Model Driven Web Engineering - MDWE*) utilizan un estilo de desarrollo en cascada mas formal; esto no significa que las metodologías ágiles no sean o no puedan ser formales pero en general, utilizan un estilo mas artesanal. Algunas de las metodologías MDWE como UWE, WebML, OOWS, OO-H y OOHDM definen un conjunto de modelos abstractos, tales como modelos de contenido, presentación y navegación, que permiten la generación de aplicaciones funcionales mediante la transformación automática (y libre de errores) de modelos.

En el caso de TDD se utilizan iteraciones que permiten agregar nuevas características a la aplicación en forma progresiva. Esta estrategia es un buen punto de partida para el proceso de desarrollo, porque los desarrolladores especifican el comportamiento esperado de los programas realizando aserciones, antes de comenzar con el desarrollo. TDD permite una mejor comunicación entre los diferentes interesados, así como las iteraciones cortas favorecen a la permanente evaluación de los requerimientos y su realización en prototipos incrementales.

La integración de características de las metodologías ágiles y las metodologías dirigidas por modelos en el proceso de desarrollo son ideas relativamente nuevas que requieren aun la definición de muchos aspectos. El método presentado en este capitulo, permite aplicar los conceptos de TDD en las metodologías MDWE. El proceso general es esencialmente el mismo que en TDD tradicional, pero en vez de escribir código, este se genera a partir de modelos de navegación y presentación utilizando una herramienta MDWE. También se generan tests automáticos y se contemplan los refactorings que puede sufrir la aplicación Web durante su desarrollo. Los test de presentación

y navegación permiten tratar la evolución de la aplicación utilizando un estilo similar a TDD. Así mismo, como en TDD tradicional, se especifica el comportamiento de la aplicación antes de su desarrollo en términos de test, utilizándolos para especificar los modelos de la aplicación, ya que estos expresan y validan la funcionalidad esperada. Algunas de las asunciones de TDD necesitan ser menos estrictas, ya que no son apropiadas para aplicaciones con un alto grado de interacción. La siguientes secciones describiremos en mayor detalle cada punto de la metodología.

5.2. Inclusión de TDD en MDWE

Como mencionamos anteriormente en TDD (capitulo 2) la nueva funcionalidad es especificada en términos de test automáticos derivados de requerimientos individuales, luego se escribe el código que permita ejecutar exitosamente los tests. Por último, se realizan los refactorings necesarios para eliminar ineficiencias como la duplicación de código.

En nuestra metodología se sigue la misma estructura pero dada la naturaleza de las aplicaciones Web, en vez de enfocarse en los test de unidad se pone énfasis en los test a nivel de interfaz e interacción. En las primeras etapas del desarrollo los test iniciales serán ejecutados utilizando *user interface mockups* y un enfoque de caja negra. La etapa de codificación de TDD se reemplaza por un paso de modelado, que luego sirve para generar el código de la aplicación mediante el uso de una herramienta MDWE. Ademas, en cada iteración es necesario agregar un paso intermedio que permita adaptar los tests, para eliminar las diferencias entre los *UI mockups* iniciales y el prototipo de aplicación generado en el paso de modelado. Cabe destacar que en esta etapa se difiere del enfoque de desarrollo dirigido por modelos convencional, ya que se trabaja a un nivel muy fino de granularidad: en el caso mas extremo, se construyen modelos para un requerimiento, generando prototipos funcionales e incrementales, donde cada requerimiento pasa a través de un paso completo de MDWE. En este sentido, la metodología propuesta es similar a los ciclos cortos de TDD tradicional, mientras que el proceso de desarrollo sigue teniendo los beneficios de trabajar con modelos.

En resumen, la metodología descripta combina las técnicas de TDD y MDWE haciendo el desarrollo Web mas *ágil*. Los pasos que componen este método son:

1. Capturar los requerimientos utilizando casos de uso o *user stories*, *user interaction diagrams* (UIDs) y *mockups* de presentación.

2. Seleccionar un caso de uso y derivar un test de interacción, que se ejecutará inicialmente contra un *UI mockup*, con el cual se especifica la navegación e interacción de la interfaz Web previo a su desarrollo.

3. Generar un prototipo funcional de la aplicación mediante la creación de modelos y generando el código en un ciclo MDWE. Se verifica la correctitud del prototipo mediante los tests definidos en el paso (2).

4. Si los test no se ejecutan exitosamente sobre el prototipo, se realizan las modificaciones correspondientes en los modelos y se regenera el prototipo hasta que los test se ejecuten de manera exitosa.

Al igual que en TDD, los pasos antes mencionados se repiten para cada uno de los casos de uso definidos, hasta que finalmente se obtiene un prototipo completo y funcional de la aplicación. La figura 5.1 muestra una visión simplificada de los pasos involucrados en el método propuesto y los contrasta con el ciclo tradicional de TDD.

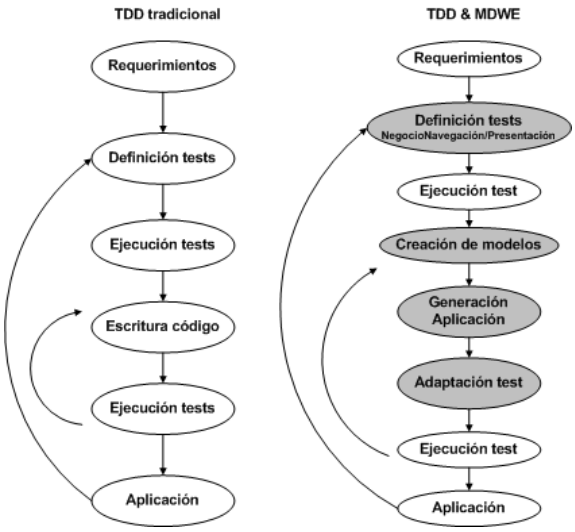

Figura 5.1: `Comparación de los ciclos de desarrollo de TDD tradicional y TDD combinado con MDWE`

A medida que la aplicación evolucione, los tests también contribuirán a verificar que la funcionalidad definida con anterioridad se preserva, luego de realizar modificaciones y *refactorings* en la navegación y presentación.

En las siguientes secciones explicaremos con mayor nivel de detalle, cada uno de los pasos involucrados en la metodología. Para ilustrar los diferentes pasos del proceso, utilizaremos como ejemplo requerimientos similares a los del sitio de venta de CDs citado en el capitulo 4.

Es importante resaltar que el método de desarrollo es independiente tanto de la metodología como herramientas MDWE utilizadas, en la derivación de los modelos; así como también del framework para la definición de los test de interfaz y navegación. A modo ilustrativo en las siguientes secciones utilizaremos la herramienta de modelado *WebRatio* [WebRatio09]: herramienta MDD (*Model Driven Development*) basada en *WebML*. Y como framework para la definición de test utilizaremos *Selenium* [Sel09]. *Selenium* es un robusto conjunto de herramientas que soportan el rápido desarrollo de test automáticos para aplicaciones Web. Provee un nutrido conjunto de funciones de test específicamente pensadas para cubrir las necesidades de testing de las aplicaciones Web. Estas operaciones son altamente flexibles, permitiendo muchas alternativas para ubicar elementos de interfaz y comparar los resultados esperados por los test contra el comportamiento actual de la aplicación.

5.2.1. Captura de requerimientos

De manera similar al enfoque MDWE, se comienza con la captura y modelado de requerimientos. En este caso, se utilizan los casos de uso o *user stories*, los diagramas UIDs y los *UI mockups*. Con estas herramientas, el analista podrá especificar los requerimientos de interfaz, navegación y negocio que la aplicación debe satisfacer. Para cada caso de uso definido, especificaremos su correspondiente diagrama UID que servirá como modelo navigacional de alto nivel y proveerá información abstracta acerca de las características de la interfaz. Tomemos como ejemplo el siguiente caso de uso:

Caso de uso - Detalle de un CD (Show CD)

Luego de realizar una búsqueda por título (ver caso de uso: búsqueda de CD por título), el usuario selecciona de la lista de CDs (resultado de la búsqueda) un CD particular para ver su información detallada. En la lista de CDs se muestra el título, compositor, imagen de la cubierta y comentario. En el detalle de cada CD, ademas de los datos antes mencionados, se podrá consultar su precio y disponibilidad.

De este caso de uso derivamos el correspondiente UID. La figura 5.2 muestra el diagrama UID derivado a partir del caso de uso.

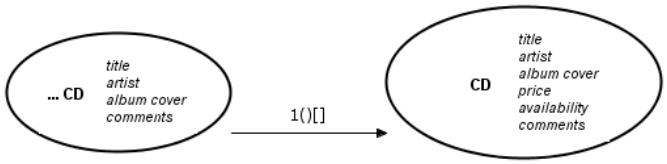

Figura 5.2: UID - Detalle de un CD

Por otra parte, mediante el uso de los *UI mockups*, se puede acordar con los clientes cual sera el aspecto visual de la aplicación antes de su desarrollo. Ademas, servirán como base para realizar los test de interfaz y navegación. A medida el desarrollo avanza dichos *mockups* pueden gradualmente convertirse en el *look and feel* final de la aplicación. La figura 5.3 muestra un *UI mockup* simplificado de la página inicial donde se listan los CDs, y la figura 5.4 muestra el *UI mockup* con el detalle de un CD en particular. Estos *UI mockups* ademas de tener los elementos básicos necesarios para cumplir con los requerimientos especificados en el caso de uso o *user story* pueden contener mas características de interfaz que sirvan para verificar funcionalidades en las próximas iteraciones. El siguiente paso es entonces definir los test de interfaz y navegación tomando como punto de partida los *UI mockups* diseñados.

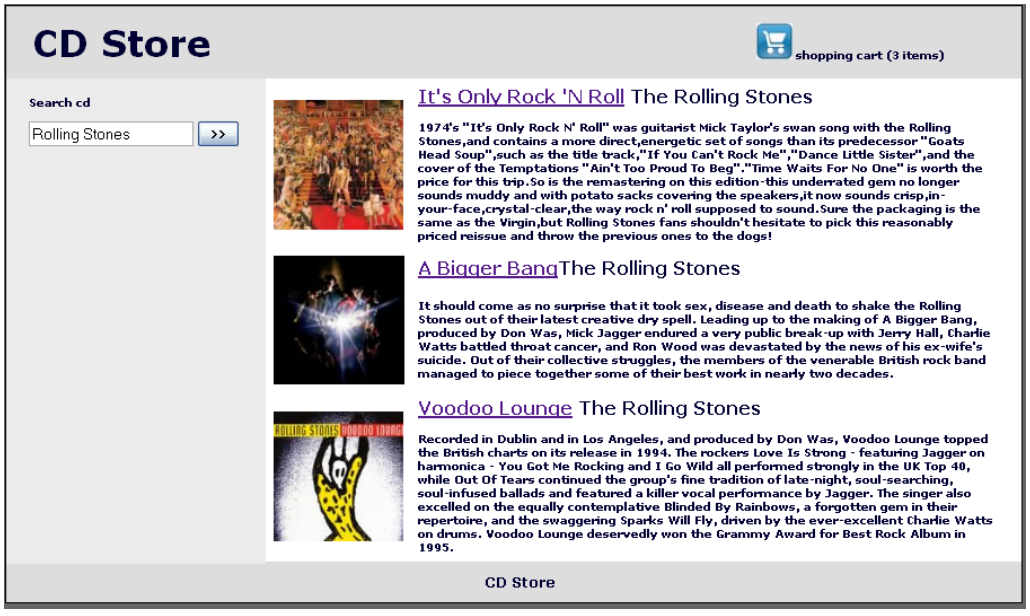

Figura 5.3: UI Mockup: listado de CDs

Figura 5.4: UI Mockup: detalle de un CD

Estos bocetos de las interfaces de la aplicación pueden ser generados utilizando algún programa para diseño. En mi caso utilice *Adobe Dreamwaver* [Adobe09], pero existen muchos otros programas exclusivos para desarrollar bocetos de las interfaces de usuario. Otra opción sería escribir directamente el código HTML con algún editor de textos, pero obviamente esto requerirá mas tiempo; y la idea es agilizar las tareas de modelado y captura de requerimientos y no poner nuevos obstáculos al desarrollo.

5.2.2. Definición de tests

Los *UI mockups* y los diagramas UIDs permiten entender el comportamiento esperado de la aplicación. Los UIDs refinan a los casos de uso mostrando como el usuario interactúa con la aplicación, mientras que los *UI mockups* complementan a los UIDs dando un ejemplo o vista preliminar del *look and feel* de la aplicación. Sin embargo, estas herramientas no pueden proveer por si mismas un elemento u objeto capaz de ser ejecutado y validado según el comportamiento esperado de la aplicación. La incorporación de los test de interacción proveen una mejor manera de validar la aplicación. Estos test son similares a los tradicionales test de unidad pero se ejecutan sobre pequeñas unidades de navegación, obtenidas de un caso de uso o *user story*, por este motivo nos referiremos a ellos como *tests de unidad de navegación*.

Los test de navegación simulan interacciones de usuario, tales como *clickear* un determinado componente de la interfaz, seleccionar un ítem de una lista desplegable, o completar una entrada de texto; y luego realizan aserciones sobre los elementos de la interfaz, tales como si determinada etiqueta *HTML* tiene un valor especifico o si se encuentra presente en la estructura DOM actual. Estos test son independientes de la herramienta MDWE utilizada ya que son ejecutados utilizando un navegador, y son adecuados para testear la lógica de negocios, navegación e interfaz de la misma manera en que es percibida por el usuario.

Es importante resaltar, que para ciertos aspectos de funcionalidad de la aplicación se siguen utilizando los test de unidad tradicionales, por ejemplo no es adecuado utilizar test de navegación

para verificar lógica de negocios realizada por el servidor de una aplicación Web. En esta caso, la metodología propuesta se mantiene inalterable y se aplican los mismos pasos utilizados en la definición de un test de navegación: especificar un test, verificar que falla, generar los correspondientes modelos y reejecutar el test para verificar que ahora es ejecutado exitosamente.

Siguiendo con nuestro ejemplo del listado de CDs, definiremos un test de unidad de navegación inicial tomando como base el UID (figura 5.2) y los *UI mockups* (figuras 5.3 y 5.4) previamente definidos. En este punto podríamos optar por definir el test en un pseudolenguaje de alto nivel o directamente expresarlo en alguna herramienta para la definición de este tipo de tests. Para este caso utilizaremos *Selenium* (aunque existen varias alternativas similares). La ventaja de la primer opción es que podríamos derivar, a partir de este metalenguaje de definición de tests, el código para un framework de test particular, tendiendo una definición mucho mas genérica y flexible, que nos permitiría cambiar de framework de test mas fácilmente.

La figura 5.5 muestra un fragmento de código para un test de unidad de navegación escrito en *Selenium*, utilizando la API de Java, que permita validar el contenido y navegación definidos en el diagrama UID y los *UI mockups* de las figuras 5.3 y 5.4).

```
  package cdstore.tests;
  import com.thoughtworks.selenium.*;

  public class CDStoreNavigationTest1 extends SeleneseTestCase {
          public void setUp() throws Exception {
                  setUp("file:///C:/Archivos de programa/EasyPHP 203.0/www/cdstore/", "*firefox");
                  selenium.setSpeed("1000");
          }

          public void testCDListToCDDetailNavigation() throws Exception {
(1)                selenium.open("cdstore-cdlist.html");
(2)                assertEquals("CD Store - CD List", selenium.getTitle());
(3)                assertEquals("The Rolling Stones", selenium.getText("//*[@id=\"artistCD2\"]"));
(4)                selenium.click("link=A Bigger Bang");
(5)                selenium.waitForPageToLoad("30000");
(6)                assertEquals("*/cdstore-cddetail.html", selenium.getLocation());
(7)                assertEquals("CD Store - CD Detail", selenium.getTitle());
(8)                assertEquals("Add to cart", selenium.getText("//*[@id=\"addtocart\"]"));
        }
}
```

Figura 5.5: `Definición de test de unidad de navegación con la API Java de Selenium`

El test comienza abriendo la página (en este caso el archivo que define el *UI mockup*) (1) y realizando una aserción de que cierto elemento en particular tiene un determinado contenido (3). Luego se realiza un *click* en un enlace de un CD (4) y se espera hasta que se haya cargado la página (5), validando la ubicación (6), y por lo tanto validando la navegación. Por último, se valida que diferentes elementos *html* contengan un texto especifico (7-8) con lo cual se válida que la interfaz haya cambiado luego de realizar el *click* en el enlace.

Aquí tenemos otra diferencia con la metodología TDD tradicional, puesto que este test no fallará la primera vez que se ejecute ya que se utilizan los *UI Mockups* que fueron definidos con anterioridad. En TDD tradicional especificamos el test previo a cualquier desarrollo, con lo cual su primera ejecución siempre es fallida. Nuestra siguiente etapa, consistirá en definir los modelos, para luego poder derivar un prototipo de nuestra aplicación y ejecutar los test que hemos definido sobre este prototipo y no sobre los *UI mockups*.

5.2.3. Derivación de modelos

Una vez que los requerimientos han sido capturados (al menos parcialmente) y se han especificado los test para un caso de uso en particular, debemos proceder a la generación de un prototipo funcional de la aplicación. Aquí es donde la metodología se diferencia de TDD tradicional por el hecho que en vez de escribir código para permitir a los test ejecutarse exitosamente se utiliza una herramienta de modelado MDWE. En nuestro ejemplo utilizaremos la herramienta de modelado *WebRatio* [WebRatio09] basada en el lenguaje *WebML* (*Web Modeling Language*). Pondremos el énfasis en los test de navegación, que son la característica distintiva de las aplicaciones Web.

Derivamos un primer modelo (en el caso de *WebML* un *data model* o modelo de datos) utilizando los UIDs como punto de partida. En dicho modelo se identifican las entidades necesarias para satisfacer las interacciones especificadas. Del diagrama UID (figura 5.2) se derivan las entidades *CD* y *Artist* y luego se mapea la navegación descripta por el diagrama UID en un diagrama Web de *WebML* mostrado en la figura 5.6.

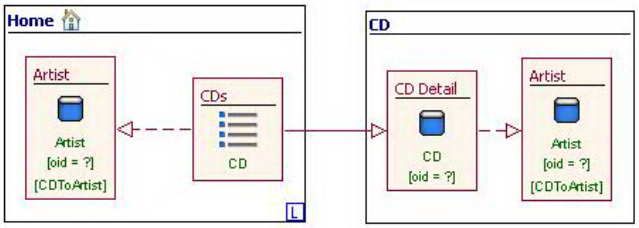

Figura 5.6: `Diagrama WebML para representar la navegación descripta en el UID`

Con la definición de este modelo, estamos en condiciones de generar el primer prototipo de nuestra aplicación. Una vez que tenemos un prototipo funcional, se deben adaptar los tests y ejecutarlos para verificar si el modelo (y por lo tanto la aplicación) satisfacen los requerimientos y no se ha corrompido la navegación planteada originalmente. Por último, se necesita ajustar la presentación de la aplicación. En el caso de WebML no se define un modelo de presentación, sino que la misma es considerada como una transformación de documentos desde una especificación de una página WebML a una página en un lenguaje especifico como JSP o ASP.NET. En otra metodología, los *UI mockups* y los UIDs también podrían ser utilizados para especificar el modelo de presentación. Como ya se cuenta con los *UI mockups* esta parte del proceso es simple, ya que solo se necesita ingresar los *UI mockups* como un *template xhtml/css* en *WebRatio*. Se deben volver a ejecutar los test para asegurar que la interacción no fue corrompida durante la modificación de los *templates*.

5.2.4. Adaptación de los test

Luego de generar los modelos, se necesita verificar que la implementación generada a partir de ellos es válida de acuerdo a la especificación de requerimientos. En particular, se desea confirmar que la lógica de negocios, navegación e interfaz es correcta con respecto a los test definidos en 5.2.2. Sin embargo, los test iniciales no serán válidos, ya que hacen referencia a los archivos *UI mockups*, y aunque la distribución de elementos sea la misma su ubicación en términos de expresiones *XPath* [XPath09] podría haber sido modificada. De todas maneras, la adaptación manual de los test es sencilla, y su mecanismo podría ser automatizado sin mayores inconvenientes. La figura 5.7 muestra como se podría adaptar el test definido en 5.2.2 para ser válido en la implementación actual.

```
package cdstore.tests;
import com.thoughtworks.selenium.*;

public class CDStoreNavigationTest1 extends SeleneseTestCase {
        public void setUp() throws Exception {
                setUp("http://localhost/cdstore/", "*firefox");
                selenium.setSpeed("1000");
        }

        public void testCDListToCDDetailNavigation() throws Exception {
(1)              selenium.open("cdstore-cdlist.jsp");
(2)              assertEquals("CD Store - CD List", selenium.getTitle());
(3)              assertEquals("The Rolling Stones", selenium.getText("//*[@id=\"page1FormBean\"]"));
(4)              selenium.click("link=A Bigger Bang");
(5)              selenium.waitForPageToLoad("30000");
(6)              assertEquals("*/cdstore-cddetail.jsp", selenium.getLocation());
(7)              assertEquals("CD Store - CD Detail", selenium.getTitle());
(8)              assertEquals("Add to cart", selenium.getText("//*[@id=\"page2FormBean\"]"));
       }
}
```

Figura 5.7: Adaptación de test de unidad de navegación para funcionar con el prototipo de aplicación generado

En primer lugar se cambia la URL inicial (notar que el prototipo generado esta implementado en JSP). Luego, como el *layout* de la lista de productos cambio debido al proceso de derivación desde *WebRatio*, las expresiones *XPath* utilizadas son inválidas ya que *WebRatio* utilizó una estructura DOM diferente. Para corregir este problema podemos utilizar una herramienta como el *XPather Plugin* [XPather09] o *Firebug* [FireB09], en donde clickeando sobre el ítem en cuestión se puede copiar y pegar la expresión *XPath* correcta. Personalmente, me pareció mucho mas completa la utilidad *Firebug*, ya que ademas de obtener expresiones *XPath* mas precisas brinda muchísimas otras utilidades para el desarrollo de aplicaciones Web. Luego de realizar todas las modificaciones, se re-ejecutan los test para verificar que ahora funcionan correctamente.

5.2.5. Nueva iteración

Una vez terminada una iteración, en donde todos los test se ejecutan correctamente, se puede comenzar un nuevo ciclo para agregar una nueva funcionalidad a la aplicación. Por ejemplo, podríamos incorporar la posibilidad de que el usuario agregue los CDs al carrito de compras. Para incorporar esta nueva característica se deben seguir los mismos pasos antes descriptos en el ejemplo previo:

1. Se modela el nuevo requerimiento con un caso de uso o *user story* y un diagrama UID.
2. De ser necesario se genera un nuevo *UI mockup*, o se extiende los que han sido previamente diseñados.
3. Se escribe un nuevo test de unidad de navegación para la funcionalidad agregada, que pueda ser ejecutado sobre el correspondiente *UI mockup*.
4. Se actualizan los modelos y se re-genera la aplicación, implementando la nueva funcionalidad para permitir la ejecución correcta del test definido.
5. Adaptar el nuevo test, para que pueda ejecutarse sobre la aplicación generada, en vez de los *UI mockups*.

6. Reejecutar todos los test para comprobar que la nueva funcionalidad ha sido incorporada correctamente. Si los test fallan se debe volver al repetir los pasos desde el paso 3.

Una vez modificados los modelos, debemos reejecutar todos los test definidos. Estos test serán aquellos que teníamos definidos hasta el momento previo de la nueva iteración (los cuales ya han sido adaptados) y los nuevos test definidos para verificar la nueva funcionalidad agregada.

5.3. Evolución de la aplicación

Las aplicaciones Web tienden a evolucionar constantemente y en periodos cortos de tiempo. Esto se debe principalmente a dos causas:

- *Nuevos requerimientos*: que se generan por pedidos de los clientes o usuarios para mejorar la funcionalidad de la aplicación. Por ejemplo, para el caso de la venta de CDs, se desea agregar la categorización de los CDs por géneros musicales, lo cual requerirá la definición de nuevos elementos del modelo (entidades, paginas, enlaces, etc).
- *Web refactorings*: se producen por el deseo de mejorar la usabilidad de la aplicación, ya sea modificando la interfaz o las características de navegación. Este tipo de cambios de modelo, son usualmente dirigidos por requerimientos no funcionales (usabilidad, accesibilidad, etc) [RRG09b].

5.3.1. Nuevos requerimientos

Luego de que la aplicación ha sido instalada (o incluso durante su desarrollo), el cliente podría desear agregar nueva funcionalidad. Por ejemplo, para el caso de nuestra tienda de venta de CDs organizar los CDs en categorías. Los nuevos requerimientos deben ser descriptos utilizando las mismas herramientas que mencionamos previamente (diagramas UIDs y *UI mockups*) y seguir el proceso de desarrollo descripto en la sección 5.2.5:

1. Agregar las etiquetas que muestren el genero/categoría del CD en los *UI mockups* del listado de CD y el detalle de un CD.
2. Agregar las aserciones a los test ya definidos para verificar la existencia y contenido de la nueva entidad de las páginas.
3. Ejecutar los test y asegurarse que fallen.
4. Modificar los modelos de dominio, navegación e interfaz (entidades, unidades y *templates* en el caso de *WebRatio*) para que reflejen el agregado del concepto de categoría de CD.
5. Generar la aplicación.
6. Ejecutar los test y readaptarlos de ser necesario. Si su ejecución falla volver al paso 4.

Luego de finalizar este ciclo, tendremos un nuevo requerimiento agregado en la aplicación y un nuevo test que valida tanto la interfaz de la lista de CDs como el detalle de un CD. Obviamente, si queremos navegar a través de las categorías el proceso es exactamente el mismo; solo debemos agregar algunos nuevos casos de uso y diagramas UIDs antes del paso 2 y definir los correspondientes test de unidad de navegación.

5.3.2. Web Refactorings

Los *refactorings* de una aplicación Web intentan mejorar la usabilidad de la misma mediante pequeños cambios del modelo, en [OGRDC08] se presenta un catalogo de los diferentes *refactorings* que se pueden realizar sobre una aplicación Web. Como ejemplo tomemos el *refactoring* de: `convertir un texto en un enlace`, que justamente permite convertir una cadena de texto en un enlace que permita dirigirnos a una nueva página (contexto) con información acerca del objeto representado por dicho texto. Supongamos en nuestro caso que deseamos convertir los nombres de los artistas en las páginas de detalle de un CD en enlaces que permitan obtener un listado de todos los álbumes del artista en cuestión [1]. De la misma manera que en los casos anteriores, debemos seguir los pasos descriptos por nuestra metodología:

1. Refactorizar el *UI Mockup* del detalle de un CD para que el texto que representa al nombre del artista sea ahora un enlace (figura 5.9).
2. Modificar el test definido para la pagina de detalle de un CD cambiando la expresión XPath para los enlaces agregados y agregar un test que valide la navegación desde el detalle de un CD a la pagina de los álbumes de un artista (figura 5.9).
3. Ejecutar los test y asegurase que fallan.
4. Modificar los correspondientes modelos *WebML* de hipertexto y presentación.
5. Derivar la aplicación
6. Ejecutar los test (adaptarlos de ser necesario). Si fallan volver al paso 4.

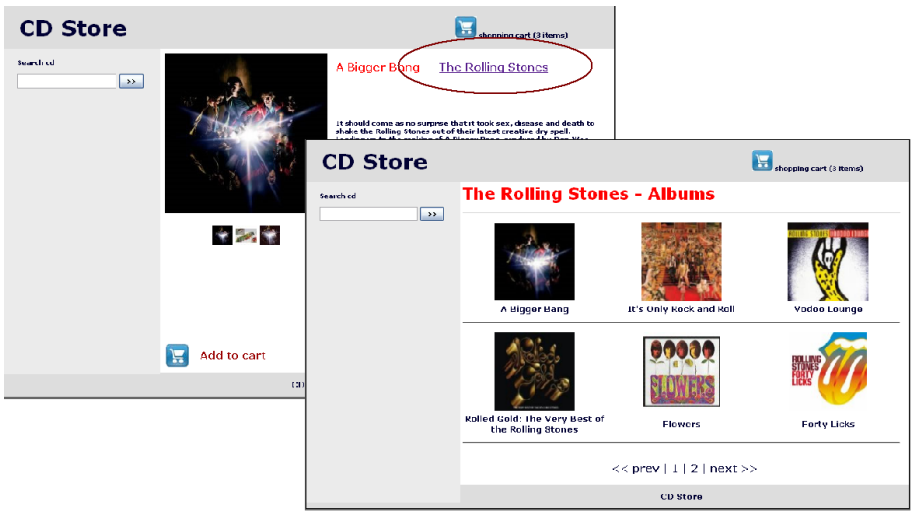

Figura 5.8: `UI Mockup refactorizado y nuevo UI mockup para mostrar los álbumes de un artista`

Al final de este ciclo habremos aplicado un Web refactoring completo a nuestra aplicación.

[1] La diferencia entre un nuevo requerimiento y un refactoring dirigido por usabilidad puede ser muy sutil, ya que estos últimos también podrían ser requeridos o generados a partir de un pedido del cliente o usuario.

```
package cdstore.tests;
import com.thoughtworks.selenium.*;

public class CDStoreNavigationTest1 extends SeleneseTestCase {
        public void setUp() throws Exception {
                setUp("http://localhost/cdstore/", "*firefox");
                selenium.setSpeed("1000");
        }

        public void testCDListToCDDetailNavigation() throws Exception {
                selenium.open("cdstore-cdlist.html");
                assertEquals("CD Store - CD List", selenium.getTitle());
                assertEquals("The Rolling Stones", selenium.getText("//*[@id=\"artistCD2\"]"));
                selenium.click("link=A Bigger Bang");
                selenium.waitForPageToLoad("30000");
                assertEquals("*/cdstore-cddetail.html", selenium.getLocation());
                assertEquals("CD Store - CD Detail", selenium.getTitle());
                assertEquals("Add to cart", selenium.getText("//*[@id=\"addtocart\"]"));
        }

        public void testCDDetailToAuthorNavigation() throws Exception {
                selenium.open("cdstore-cddetail-artistlink.html");
                assertEquals("CD Store - CD Detail", selenium.getTitle());
                selenium.click("link=The Rolling Stones");
                selenium.waitForPageToLoad("30000");
                assertEquals("*/cdstore-artistalbums.html", selenium.getLocation());
                assertEquals("The Rolling Stones - Albums", selenium.getText("//*[@id=\"artist_album_title\"]"));
                assertEquals("CD Store - Albums by Artist", selenium.getTitle());
        }

}
```

Figura 5.9: Agregado de nuevo test de navegación para verificar el Web Refactoring realizado en la aplicación

5.4. Generación y adaptación automática de los tests

Dos de las tareas descriptas en este proceso de desarrollo pueden ser potencialmente automatizadas. En particular, se podría automatizar la derivación de los test a partir de los diagramas UIDs y la adaptación de los test producto de los cambios producidos por los refactorings. A continuación explicaremos brevemente cada caso.

5.4.1. Derivación de los test

De lo visto anteriormente observamos que los test de navegación que hemos escrito, a lo largo de este capitulo, tienen una relación directa con los diagramas UIDs. Mas concretamente, los test de navegación deberían tener tantos cambios de contexto (redirección de una nueva página) como diferentes estados tenga el diagrama UID en cuestión. Teniendo esto en mente podríamos derivar un *template* para el código de nuestro test de navegación a partir de un diagrama UID. En primer lugar, necesitaremos representar nuestro diagrama en un formato fácilmente interpretable, como por ejemplo XML; de hecho el editor de diagramas UID-RIA guarda los diagramas con este formato. En este punto estaríamos en condiciones de implementar una herramienta que tome como entrada un diagrama en formato XML y nos de como salida el código para la definición del test. Básicamente, esta herramienta realizará dos tareas principales:

1. Analizar el diagrama y obtener todos los caminos de navegación válidos en base a la cantidad de estados definidos y sus interconexiones.

2. Para cada camino de navegación válido: definir la acción para cambiar de contexto (en el caso de Selenium, clickear un link y esperar por la carga de la nueva página) y agregar las aserciones mínimas para verificar que se esta en el contexto correcto (por ejemplo, aserción acerca del título de cada página).

Una vez generada la estructura básica del test, podemos mejorar el test agregando aserciones mas complejas que validen la estructura y contenido de las páginas. Para lograr una automatización completa del proceso, necesitaremos proveer mas información de entrada a los diagramas, como por ejemplo asociar el correspondiente *UI mockup* a cada estado de interacción del diagrama.

Otra alternativa para facilitar la definición de los test de navegación es la utilización de alguna herramienta gráfica. Como ejemplo podemos nombrar al *plugin* gráfico para *Eclipse CubicTest* [CubicT10]. Este *plugin* desarrollado con *Eclipse GMF* nos permite definir los test de manera gráfica y luego generar código para las plataformas de testing *Selenium* y *Watir* [Watir10]. La imagen 5.10 muestra como podemos definir un test en *CubicTest* simplemente componiendo y relacionando las diferentes entidades gráficas.

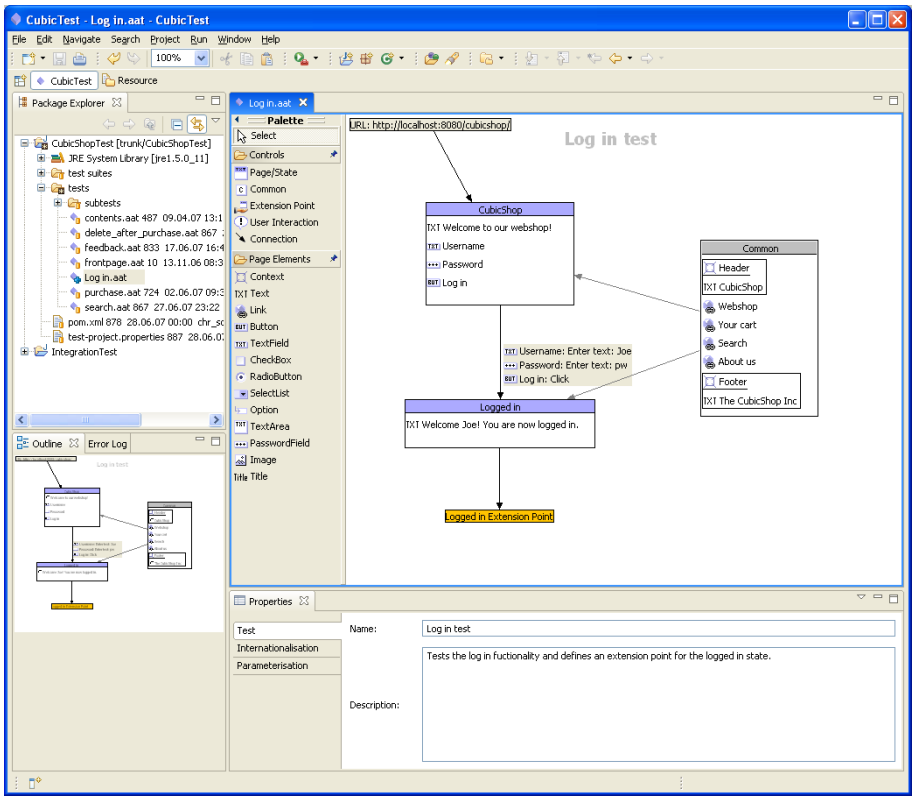

Figura 5.10: `Definición de un test de navegación mediante CubicTest`

5.4.2. Adaptación de los test luego de un refactoring Web

Durante el ciclo de desarrollo, los test que fueron definidos en las primeras iteraciones deberían siempre ejecutarse correctamente (a excepción que alguno de los requerimientos ya procesados haya cambiado de forma dramática). Sin embargo, los Web *refactorings* presentan un nuevo desafío: aun no siendo originados por un nuevo requerimiento, pueden hacer que un test de navegación falle, ya que podrían (sutilmente) cambiar la estructura navigacional y/o la interfaz de la aplicación. En otra palabras, los test deben ser adaptados para seguir siendo útiles luego de un *refactoring*.

Los *refactorings Web* pueden ser catalogados, de la misma manera que los patrones de diseño. Por lo tanto, es posible automatizar el proceso de transformación de los test. Esta transformación dirigida por *refactorings* de los test debe ser realizada luego de que los *UI mockups* y diagramas UIDs hayan sido modificados para reflejar el nuevo comportamiento esperado. Para transformar un test necesitamos seguir los siguientes pasos:

1. Seleccionar la trasformación del test asociada con el *refactoring* del catalogo.
2. Configurar la transformación del test con los UIDs, *UI mockups*, ubicación de los tests y los parámetros específicos de la transformación.
3. Aplicar la transformación del test.

Para lograr la automatización de la transformación de un test, en primer lugar debemos abstraer los conceptos involucrados en un test Web. Un test Web es una secuencia de interacciones y aserciones que tienen por objetivo validar el comportamiento de una aplicación. Por ejemplo: clickear un enlace, clickear un botón, tipear un texto en una entrada de texto, etc. Las aserciones, por su parte permiten asegurar que un predicado es valido en el contexto actual. Podemos realizar muchas aserciones diferentes sobre una pagina Web tal como *assertTitle*, *assertTextPresent*, etc. De esta manera, podemos abstraer un test Web por medio de un modelo que a su vez abstraiga cada uno de estos conceptos.

Los test individuales pueden ser abstraídos, desde su código fuente a una instancia del modelo definido, mediante la utilización de un *parser*. Una vez que el test es mapeado a un conjunto de objetos, estos pueden ser fácilmente manipulados. Por ejemplo, para agregar una aserción de titulo basta con crear una nueva instancia de la clase *AssertTitle* y agregarla a la instancia de la clase *WebTest*. La transformación de un test es diseñada y codificada con objetos, con lo cual el algoritmo para realizar la transformación puede ser codificado y encapsulado en una clase. Una vez que la transformación del test ha sido aplicada, debemos volver a traducir los objetos nuevamente a código; seguramente con la asistencia de un algoritmo *pretty print*.

Como ejemplo tomemos el algoritmo para convertir un texto en un enlace; la transformación del test puede resumirse en los siguiente pasos:

1. Obtener la ubicación del test.
2. Obtener la ubicación del texto a remplazar.
3. Cambiar la ubicación del la instancia *AssertText* del texto. Si no existía dicha aserción, crear una nueva instancia de la clase *AssertText*.
4. Crear una nueva instancia de un *WebTest*. Crear un instancia de *OpenURL* (que referencie al correspondiente *UI mockup*) y clonar la instancia de *AssertText* del paso 3. Agregar ambas instancias al nuevo *WebTest*.
5. Crear instancias de las clases *Click* y *Wait*, referenciarlas a la ubicación del nuevo enlace, y agregarlas a la instancia del nuevo *WebTest*.
6. Obtener la ubicación esperada y el texto que identifica al nuevo contexto.
7. Crear instancias de las clases *AssertText* y *AssertLocation* con los correspondientes valores obtenidos en el paso 6.

El resultado de aplicar este algoritmo se verá similar al obtenido en la sección 5.3.2. De esta manera, mediante este enfoque podemos automatizar el proceso de transformación de los Web test en base al catalogo de *Web refactorings* que deseemos aplicar.

5.5. Resumen

En este capitulo se describió una metodología de desarrollo Web que combina el estilo ágil, iterativo e incremental de TDD con las metodologías de ingeniería Web dirigidas por modelos MDWE.

El proceso de desarrollo propuesto es independiente tanto de la metodología y herramientas MDWE, como del framework para la definición de test utilizados. Permitiendo que se pueda optar por las herramientas con las cuales los desarrolladores y diseñadores están mas familiarizados o tienen mayor experiencia.

Los pasos que componen la metodología son:

- Captura de requerimientos mediante casos de uso o *user stories*, diagramas UIDs y *UI mock-ups*.
- Definición de uno o varios test de navegación para el requerimiento que esta siendo modelado.
- Definición de modelos utilizando alguna herramienta MDWE, que permita generar un prototipo sobre el cual ejecutar los test definidos.
- Adaptación de los test para poder ejecutarlos sobre los sucesivos prototipos generados.
- Re-ejecución de los test para validar que la aplicación cumple con los requerimientos especificados, con cada iteración del ciclo de desarrollo.

Este ciclo se repite para cada requerimiento, con lo cual permite obtener en forma progresiva un prototipo de aplicación cada vez mas completo y funcional y eventualmente la aplicación final.

Ademas, el proceso de desarrollo permite tratar con la evolución de la aplicación contemplado el surgimiento de nuevos requerimientos que necesiten ser validados así como las modificaciones debido a los Web refactorings.

Tanto la derivación inicial de los test a partir de los diagramas UIDs como la adaptación de los test, para que puedan ser ejecutados sobre los prototipos derivados con la metodología MDWE utilizada, son tareas que pueden ser automatizadas total o parcialmente mediante el uso heurísticas y reglas de derivación.

Capítulo 6

MDWE y TDD en el desarrollo de aplicaciones RIA

En el capitulo 5 describimos una metodología que nos permite combinar las técnicas de TDD y MDWE en el desarrollo de aplicaciones Web. En este capitulo describiremos todos los puntos a tener en cuenta y las modificaciones necesarias sobre la metodología de desarrollo planteada para que la misma sea aplicable al desarrollo de aplicaciones RIA.

6.1. Introducción

Existen tres puntos principales de nuestra metodología de desarrollo en los cuales deberemos tener en cuenta varios detalles, si queremos aplicarla al desarrollo de aplicaciones RIA:

- `Metodología y herramientas MDWE utilizadas`: si bien destacamos que el método de desarrollo es independiente tanto de la metodología como de las herramientas MDWE utilizadas, seria deseable que la metodología MDWE empleada nos provea el soporte para modelar y derivar aspectos RIA que no están presentes en una aplicación Web estándar.
- `Framework para la definición de test`: al seleccionar un framework para implementar los test de navegación, deberemos buscar uno que permita la interacción con la tecnología de desarrollo seleccionada para implementar la aplicación RIA. Por ejemplo, no podemos utilizar *Selenium* para testear una aplicación desarrollada en Flash.
- `Captura de requerimientos y comportamiento RIA`: debemos analizar cual de nuestras herramientas para capturar los requerimientos (casos de uso, diagramas UIDs y *UI mockups*) es la mas adecuada para capturar y reflejar comportamientos RIA. Puede que sea necesario utilizar herramientas adicionales para la captura de los requerimientos.

En lo que resta de este capitulo trataremos en mayor detalle cada una de estas cuestiones.

6.1.1. Metodologías MDWE en Aplicaciones RIA

Si bien nuestra metodología de desarrollo, es independiente de la metodología MDWE utilizada, la modelización de aplicaciones RIA involucra nuevos conceptos que no están presentes o no tienen demasiada relevancia en el modelado de aplicaciones Web tradicionales. Por ejemplo, el modelo

navigacional de una aplicación RIA diferirá en gran medida de la versión Web tradicional de la misma aplicación, ya que muchos eventos sucederán en el mismo contexto (continuidad visual).

El diseño de aplicaciones RIA con metodologías MDWE requiere la adaptación del flujo de desarrollo Web de las aplicaciones Web tradicionales para considerar las nuevas capacidades del lado del cliente, las nuevas características de presentación y los diferentes flujos de comunicaciones entre el cliente y el servidor.

En una comparativa, realizada en [PLCS05], entre las capacidades para modelar características RIA de las diferentes metodologías MDWE existentes se concluye que ninguna de ellas permite cubrir todos los aspectos relacionados con este tipo de aplicaciones. También se afirma que de todas las metodologías existentes; las mas recientes tales como OOHDM, OO-H, UWE, WebML, y W2000; aparentan ser mas flexibles para soportar el agregado de extensiones que permitan modelar los aspectos concernientes a RIA.

Ademas, no podemos omitir el hecho de que las herramientas mas modernas de desarrollo RIA son de bajo nivel, dependientes de la tecnología, orientadas a código, y no proveen ninguna funcionalidad para el modelado independiente de la plataforma y la generación de código.

Por este motivo, nuestro primer punto a resolver será utilizar una metodología MDWE que nos permita modelar ciertos aspectos y características propias de una aplicación RIA. Ya que en nuestro caso utilizamos *WebRatio* como prueba de concepto, en el apéndice B sección describiremos brevemente una ampliación del lenguaje *WebML* para el modelado de los conceptos involucrados en las aplicaciones RIA. En el caso de *WebRatio*, hay que destacar que su ultima versión tiene soporte para generar aplicaciones con funcionalidad AJAX, entre algunas de las funcionalidades que provee podemos nombrar:

- Drag and drop
- Actualización de pagina selectiva
- Selección en cascada
- Actualización periódica de paginas
- Administración dinámica de eventos en formularios
- Tool tips
- Auto completado de campos

Varias de estas características son los mismos patrones de diseño que describiremos en el capitulo 7. Se puede ver una demostración de algunas de estas características en [WRAjax10], la figura 6.1 muestra la implementación de *tool tips*, generados con *WebRatio* en *AJAX*.

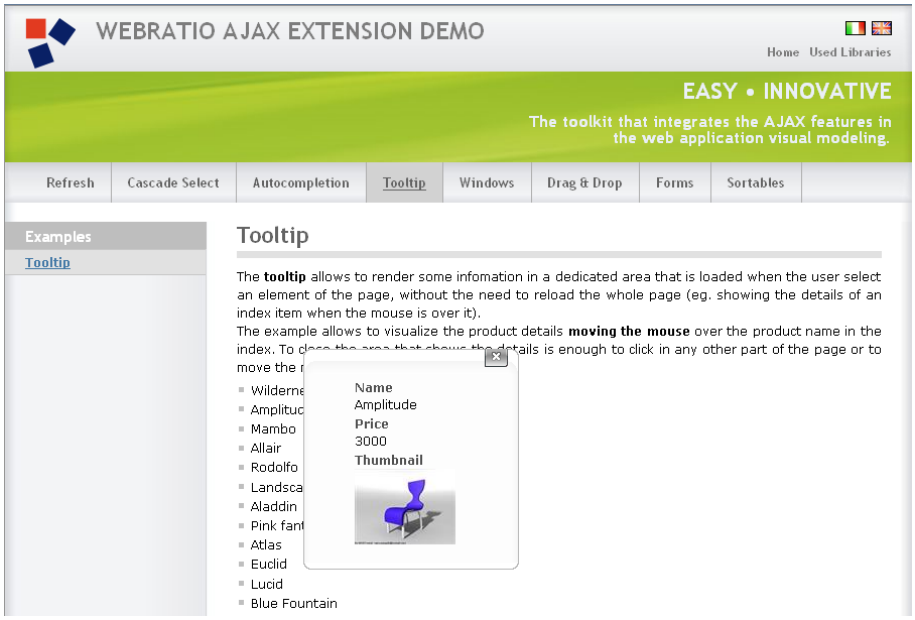

Figura 6.1: `Ejemplo de tooltip implementado con AJAX y derivado con WebRatio`

6.1.2. Framework para la definición de test de navegación

Otro punto a tener en cuenta es la selección del framework para la definición de los tests de navegación. En el capitulo 5 se utilizo el framework *Selenium* [Sel09] para la especificación de los test de unidad de navegación. *Selenium* permite realizar diferentes acciones y aserciones sobre aplicaciones Web basadas en una estructura *html*.

De esta manera, en el caso de desarrollar una aplicación RIA híbrida basada en AJAX y *html*, el framework *Selenium* será adecuado para la definición de los test. Sin embargo, si la aplicación RIA a desarrollar es implementada con otra tecnología deberemos buscar un framework de testing que permita interactuar con la interfaz de la aplicación desarrollada. Por ejemplo, en el caso de una aplicación desarrollada con *Adobe Flash* no dispondremos de una estructura *DOM html* para acceder a los diferentes *widgets* de la interfaz.

En el caso de *Adobe Flash*, *Adobe Flex* y *Microsoft Silverlight* existen extensiones del *framework Selenium* para automatizar los test de interfaz y navegación. El framework que extiende las capacidades para interactuar con aplicaciones *Adobe Flash* es *Flash-Selenium* [SelFla09], para aplicaciones *Adobe Flex* es *Selenium-Flex API* [SelFle09] y para aplicaciones *Microsoft Silverlight* es *Silverlight-Selenium* [SelSil09].

Existen varios framework alternativos a *Selenium* para automatizar los test de interfaz. Como ejemplos podemos nombrar a *RIATest* [RIATest09], que permite testear aplicaciones *Adobe Flex*; y *Squish* [Squish09] que permite automatizar los test de aplicaciones basadas en *html* y *AJAX*. Si bien ambos *frameworks* brindan características muy similares a las de *Selenium* la principal diferencia con este ultimo es que ambas son herramientas bajo licencia comercial.

6.2. Especificación de los requerimientos RIA

El ultimo punto que debemos tener en cuenta para adaptar la metodología de desarrollo y utilizarla en aplicaciones RIA es como especificaremos los requerimiento RIA con las herramientas para capturar requerimientos disponibles.

Recordemos del capitulo 5 que contamos con tres herramientas: 1) casos de uso, 2) diagramas UIDs y 3) *UI mockups*. Obviamente la primer herramienta que descartamos son los caso de uso, puesto que escribir frases como: `El usuario ingresa el titulo del CD a buscar, cuya entrada tiene la característica RIA de autocompletado` complicaría demasiado su correcta interpretación por parte de los diferentes interesados. De esta manera, nos quedan los diagramas UIDs y los *UI Mockups* para asistirnos en la captura de los requerimientos RIA.

En particular, utilizaremos los diagramas UIDs para reflejar algunos comportamientos RIA. Obviamente, al diseñar *UI mockups*, para acordar el *lock and feel* de la aplicación, también podremos incluir estas características. Por ejemplo, si una entrada de texto tiene la característica RIA de autocompletado, el *UI mockup* podría tener algunos valores pre-cargados para simular dicho comportamiento al ingresar un texto en la entrada en cuestión.

El problema al que nos enfrentamos ahora es como especificar estos comportamientos en los diagramas UIDs sin complicar su notación. Recordemos que una de las principales ventajas de estos diagramas es su notación simple que permite su fácil interpretación por parte de los usuarios. Si agregamos nuevas entidades gráficas para cada requerimiento RIA que quisiéramos modelar sobre-cargaríamos la notación, convirtiéndola en una herramienta poco útil para la captura de requerimientos.

Teniendo en mente este problema, lo ideal es entonces especificar los requerimientos RIA mediante un nota adjunta al componente gráfico involucrado. Esta alternativa tiene sentido si no disponemos de ninguna herramienta digital para dibujarlos. Es decir, si queremos especificar los diagramas UIDs en papel. La figura 6.2 muestra la representación gráfica de un diagrama UID con un solo estado donde el usuario debe ingresar un título y una dirección de e-mail. Aquí se desea reflejar en los requerimientos que la entrada correspondiente al titulo posee la característica RIA de auto-completado, mientras que la dirección de *e-mail* ingresada tiene validación en línea (se válida que la dirección tenga el formato de una dirección de correo electrónico, antes de ser enviada).

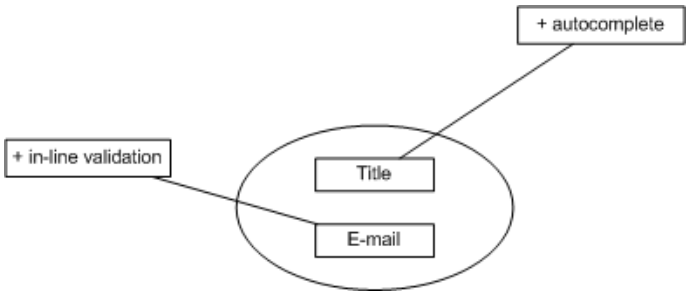

Figura 6.2: `Posible forma de especificar requerimientos RIA en un diagrama UID`

En nuestro caso disponemos del editor de diagramas UIDs implementado en Eclipse GMF [GMF09], con lo cual podemos reemplazar a las notas adjuntas seleccionado el componente al que queremos especificarle una característica RIA y mediante el inspector de propiedades setear el valor adecuado para la propiedad. Ademas, para diferenciar un componente normal de uno que posee alguna característica RIA se utiliza un icono indicador que se muestra según haya o no alguna de estas propiedades seteadas. La figura 6.3 muestra el mismo fragmento de diagrama UID especificado anteriormente representado en el editor UID-RIA.

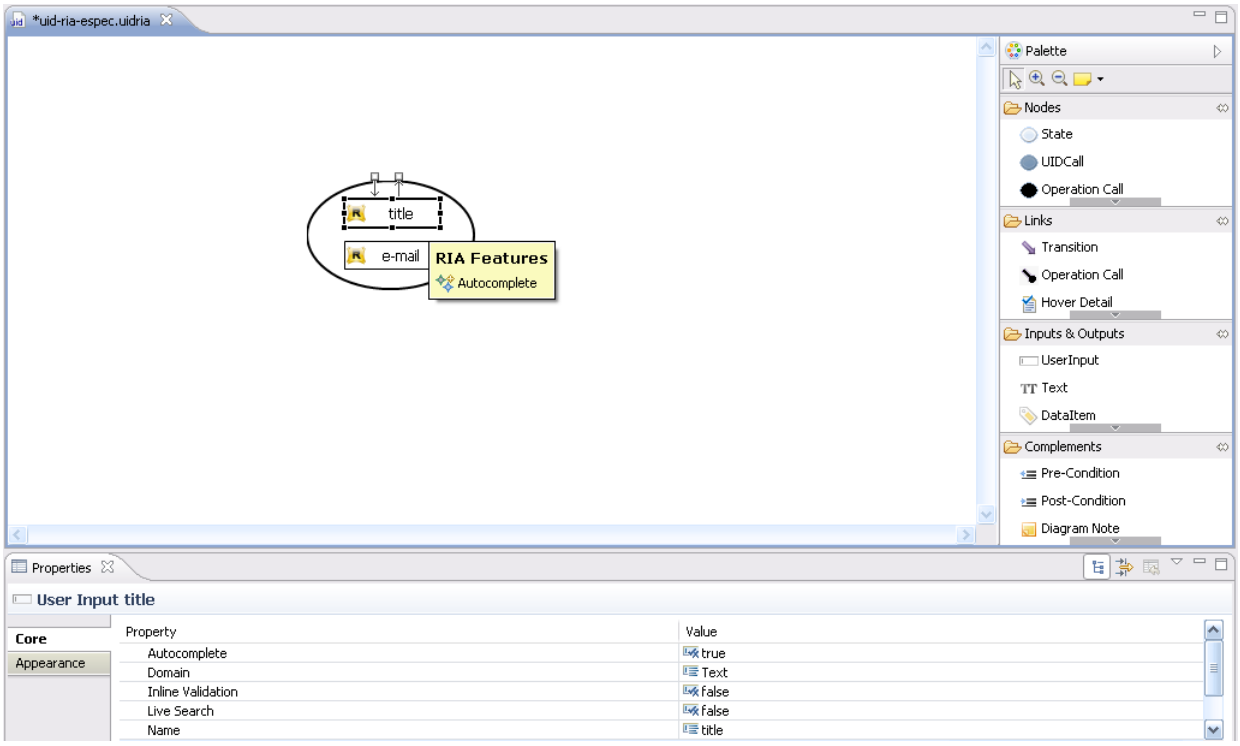

Figura 6.3: `Especificación de diagramas UID con funcionalidad RIA mediante el editor UD-RIA`

Al seleccionar un componente se puede setear sus propiedades relacionadas con aplicaciones RIA, luego al pasar con el *mouse* sobre un componente se mostraran automáticamente todas las propiedades RIA que tiene activadas.

Obviamente esta es una implementación parcial, que nos permite especificar algunos pocos comportamientos RIA. Veremos ejemplos concretos de estos comportamientos en el siguiente capitulo.

6.3. Resumen

Adaptar nuestra metodología de desarrollo involucra tener en cuenta varios aspectos:

- `Metodología MDWE`: necesitamos ampliar alguna metodología MDWE para soportar el modelado RIA o investigar la creación de nuevas metodologías para tal cuestión. Tal vez, este sea el punto donde hay muy poca investigación y propuestas realizadas. Esto se debe a que tanto las metodologías MDWE como las aplicaciones RIA son conceptos relativamente recientes y mas aun la combinación de ambos.

- `Framework para test de interfaz y navegación`: de acuerdo a nuestra tecnología de desarrollo RIA, deberemos encontrar un framework que permita interactuar con la misma, de manera similar a las capacidades que provee el framework *Selenium* para testear aplicaciones basadas en HTML y Ajax.

- `Captura de requerimientos RIA`: queremos poder especificar características RIA desde la fase de captura de requerimientos. Para esto podemos ampliar nuestros diagramas UIDs y *UI Mockups* e investigar la inclusión de otras herramientas para la captura de este tipo de requerimientos, mas relacionados a requerimientos de usabilidad que de funcionalidad.

Capítulo 7

Definición de tests para requerimientos RIA

Como prueba de concepto de nuestra metodología de desarrollo (5) y su adaptación para soportar el modelado de los conceptos concernientes a aplicaciones RIA (capitulo 6); ampliaremos los test de navegación para que sean capaces de verificar el correcto funcionamiento de algunas características RIA, ademas de seguir siendo útiles para verificar la navegación y funcionalidad de la interfaz.

7.1. Introducción

Las características a las que nos referimos están basadas en patrones de diseño extraídos principalmente de la librería *Yahoo! Design Pattern Library* [YDPL09]. Es importante destacar que existen muchos catálogos de patrones similares, tales como [Quince09], [IntP09], [UIPF09], [Welie09] y [AjaxP09]. Varios de estos patrones son comunes en mas de uno de estos catálogos o librerías, en ocasiones descriptos bajo nombres diferentes (por ejemplo: *Hover Tooltip* o *Tooltip Invitation*).

De todos estos patrones hemos seleccionado algunos para agregarlos como requerimientos de usabilidad en nuestra aplicación en desarrollo y definiremos uno o varios test para verificar su funcionamiento. Para cada patrón de diseño analizado abordaremos los siguientes puntos:

- Describiremos el patrón de diseño y citaremos algunos ejemplos reales de su utilización.
- Utilizaremos nuestros *UI mockups* de la misma manera que se describió en el capitulo 5, pero les incorporaremos la simulación de la funcionalidad RIA a modelar. A tal fin utilizaremos la librería *Yahoo! User Interface Library (YUI)* [YUI09], que es la implementación mediante la tecnología *AJAX + JavaScripts* de los patrones descriptos por *Yahoo! Design Pattern Library*. De esta manera, también podremos validar con los interesados el *look and feel* y funcionalidad que incorporan estas nuevas características en la aplicación.
- Utilizaremos el editor de diagramas RIA-UID para especificar en los requerimientos la utilización del patrón. Es decir, indicaremos dentro de un estado de interacción cual o cuales componentes tiene alguna característica RIA definida. De esta manera, podremos derivar un test de navegación completo, de la misma manera que describimos en el capitulo 5, pero *inyectando*(entre medio de la simulación de navegaciones y aserciones acerca del contenido de la interfaz) el código que verifique el correcto funcionamiento del patrón de diseño especificado.

Para simplificar los ejemplos, supondremos el desarrollo y generación de una aplicación híbrida compuesta de *templates* de páginas Web convencionales (HTML, AJAX y Javascript) con componentes RIA, con lo cual podemos utilizar el *framework Selenium* para la definición de los test. Sin embargo, las ideas que describiremos no están supeditadas a una tecnología en particular. Podríamos fácilmente cambiar nuestra tecnología de desarrollo RIA, suponiendo que contamos de una herramienta MDWE capaz de derivar aplicaciones RIA y un framework de testing capaz de interactuar con las interfaces de una aplicación desarrollada en esa tecnología.

También es importante destacar que los test definidos podrían ser utilizados fuera de la metodología propuesta en el capitulo 5. Es decir, implementar la aplicación simplemente escribiendo código en el *framework* RIA elegido y luego definir los test para testear la navegación, interacción y los patrones de diseño. De todas maneras, nuestra intención es utilizar esta ampliación de los test de navegación como parte de la metodología TDD + MDWE.

Por último, y no menos relevante, debemos mencionar que la idea de especificar requerimientos RIA mediante el uso de diagramas UIDs y *UI Mockups*, para luego derivar test de navegación e interacción que verifiquen su funcionamiento, es solo una de las alternativas para ampliar nuestra metodología de desarrollo al mundo de las aplicaciones *Web 2.0*. Por ejemplo, en [FTSF09] se describe como modelar este tipo de patrones directamente utilizando WebML. Esto significa que podemos investigar diferentes alternativas para especificar los requerimientos RIA en el proceso de desarrollo de una aplicación.

7.2. Definición de patrón de diseño

Antes de pasar a los ejemplos concretos comencemos por definir a que nos referimos con patrón de diseño. Podemos definir a un patrón de diseño como:

> *Un problema recurrente en el diseño de software y una solución base general y reutilizable para dicho problema.*

Otra definición similar sería:

> *Un patrón describe una solución optima para un problema común dentro de un contexto especifico.*

Cada patrón esta compuesto de 4 componentes primarios:

1. Título
2. Descripción del problema
3. Un contexto
4. Una solución

Ya que una imagen vale mas que mil palabras, la descripción de cualquier patrón no estaría completa sin ejemplos concretos que lo representen visualmente. La justificación y los aspectos referidos a la accesibilidad también pueden ser mencionados como parte de la descripción del patrón.

En lo que resta de este capitulo describiremos algunos de estos patrones de diseño y para cada uno abordaremos los puntos antes mencionados.

7.3. Autocomplete

La inclusión de la característica de auto-completado (*autocomplete*), asociada a una entrada de texto estándar, permite al usuario ser mas rápido y preciso cuando la información requerida es ambigua, difícil de recordar o proviene de una base de datos extensa donde el numero total de ítems supera a la longitud máxima razonable de una lista desplegable (mostrar una cantidad excesiva de ítems para seleccionar puede resultar confuso e incomodo para el usuario). El autocompletado permite que las entradas de usuario se puedan completar mas rápidamente eliminando la ambigüedad acerca de la entrada de datos esperada, evita además los errores de tipeo reduciendo la cantidad de opciones seleccionables.

7.3.1. Descripción del patrón

Alias

- *Autocompletion, Continous Filter, Entry Suggestions, Live Search*

Descripción del problema

- El usuario necesita ingresar un ítem en una entrada de texto donde el dato a ingresar es ambiguo, difícil de recordar o proviene de un conjunto de ítems muy extenso, lo cual aumenta la probabilidad de ser ingresado erróneamente.

Cuando utilizarlo

- Los ítems sugeridos pueden ser obtenidos de un conjunto de datos manejable.
- El ítem de entrada puede ser ingresado por varias alternativas (mouse, tecla TAB, selección).
- El ítem de entrada coincide con algún ítem de datos especifico del sistema.
- Para lograr que la entrada de datos pueda ser completada, rápida y correctamente.
- Existe un conjunto limitado de respuestas que son válidas, pero el número total de ítems es demasiado grande o inconveniente para mostrarse en una lista desplegable estándar.
- Es posible predecir o adivinar la respuesta del usuario.
- Las posibles respuestas son largas y/o difíciles de recordar o tipear (por ejemplo, URLs y códigos de aeropuertos).
- Las respuestas pueden ser tipeadas de maneras diferentes (por ejemplo, San Francisco International Airport o SFO).
- Es importante la precisión de la respuesta.

Solución

- Layout
  - Utilizar una entrada de texto estándar.
  - Etiquetar apropiadamente la entrada de texto para que refleje las expectativas del usuario acerca de cuales serán los valores sobre los que se realizará la búsqueda de sugerencias.

- Interacción
  - A medida que el usuario tipea en la entrada de texto, se muestra una lista de ítems sugeridos que tienen mayor coincidencia con la entrada ingresada. La lista de ítems sugeridos se reduce a medida que el usuario refina la expresión ingresada.
  - Los ítems sugeridos son mostrados en una lista desplegable debajo de la entrada de texto. Los ítems pueden estar basados en el conjunto de datos completo o en otro criterio mas preciso como la frecuencia de uso.
  - Se pueden mostrar varios campos de información para cada ítem sugerido.
  - Resaltar el ítem con mayor coincidencia:
    - Mostrar como primera opción el ítem con mayor coincidencia en la lista.
    - Resaltar el color de fondo del ítem.
  - Para cada ítem sugerido, resaltar los caracteres (generalmente poniendo el texto en negrita) que coinciden con la entrada actual del usuario.
  - Permitir al usuario borrar un carácter de la entrada y volver a mostrar los ítems previamente sugeridos.
  - Permitir al usuario completar la entrada de texto presionando las teclas *Tab* o *Enter*. Esta acción hará que el ítem con mayor coincidencia sea aceptado como el valor de la entrada de texto.
  - Permitir seleccionar un ítem en particular de la lista de sugerencias mediante el cursor del mouse o utilizando las teclas de dirección (up y down).
  - Ocultar la lista de ítems sugeridos mediante la tecla *Esc*. La lista de sugerencias es ocultada y se restaura el auto-completado si el usuario vuelve a tipear.
  - Si la entrada de texto permite múltiples campos de información, permitir al usuario utilizar la coma como separador de cada campo.
    - Ingresando una coma se completa un campo en particular, y al ingresar nuevos caracteres la sugerencias son basadas en el siguiente campo de información.

Ventajas

- Reduce el número de caracteres tipeados por el usuario permitiendole completar la entrada de texto con mayor rapidez.
- Remueve la ambigüedad mostrando información adicional de formateo en la lista de ítems sugeridos (por ejemplo direcciones de e-mail).
- El *feedback* continuo ayuda al usuario a encontrar rápidamente la respuesta adecuada.

Implementación

- Hay varios enfoques técnicos para implementar *autocomplete*. Cuando el conjunto de sugerencias no es muy extenso (por ejemplo destinos de vuelo) el conjunto completo de ítems puede ser embebido en el código de la página Web. Pero para conjuntos muy grandes (como el ejemplo de Google, ver siguiente sección) la solución mas usual es la comunicación con algún servidor para recuperar las sugerencias (por ejemplo, mediante la tecnología AJAX).

Ejemplos

- Esta patrón es muy común en aplicaciones RIA. Como ejemplo podemos citar a los correos electrónicos *Gmail* [Gmail09] y *Yahoo Mail* [YMail09] (figura 7.1), donde al ingresar el destinatario de un nuevo mensaje se despliega la lista de contactos permitiendo acelerar la búsqueda e ingreso de la dirección de correo deseada. Otro ejemplo muy conocido es el buscador *Google* [Goo09] mostrado en la figura 7.2.

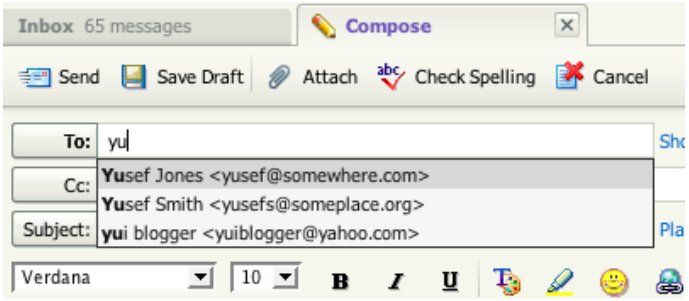

Figura 7.1: `Autocomplete: en Yahoo Mail, a medida que el usuario ingresa caracteres se muestran las sugerencias que coinciden con la entrada`

Figura 7.2: `Autocomplete :en el buscador Web de Google, para cada término de búsqueda sugerido se muestra la cantidad de resultados que existen al buscar por dicho término`

7.3.2. Definición de test de navegación e interacción

Tomemos como ejemplo el diagrama UID: *Selección de un CD en base al nombre del compositor* (figura 4.3) visto en el capitulo 4. Como primer paso queremos dejar plasmado en los requerimientos que la entrada de datos referente al compositor *artist name* tendrá el patrón de diseño RIA de *autocomplete*. Utilizando el editor implementado definimos el diagrama UID y seteamos la propiedad RIA *autocomplete* en el componente gráfico (Entrada de usuario) que representa el nombre del compositor. La figura 7.3 muestra la nueva versión del diagrama UID.

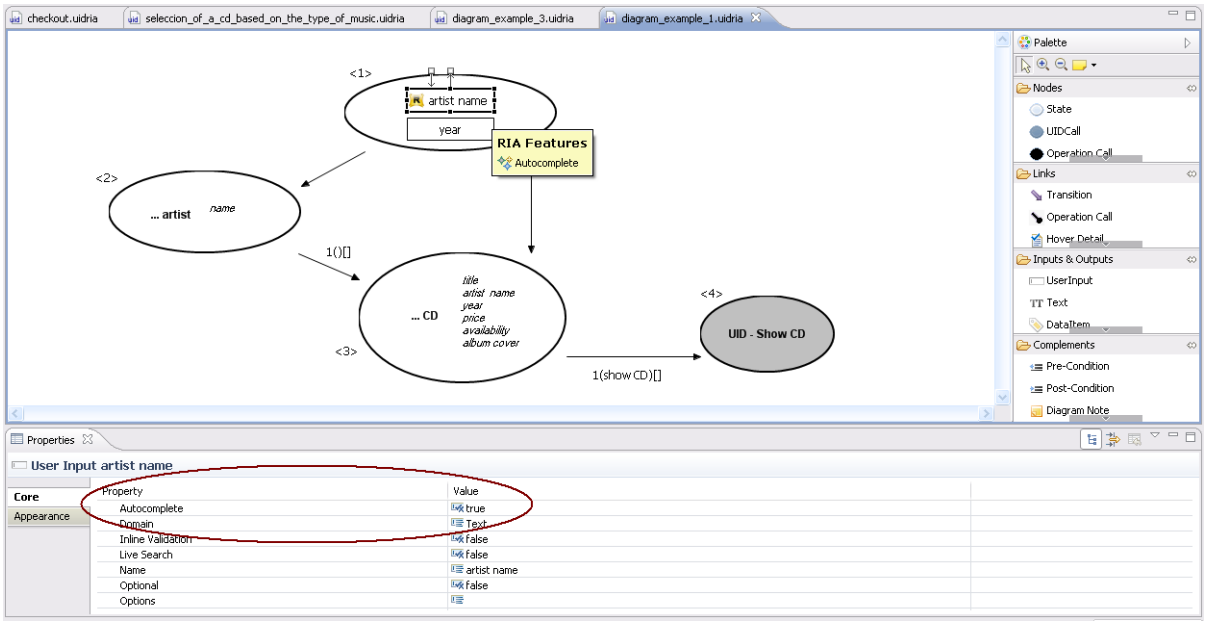

Figura 7.3: `Especificación del patrón RIA Autocomplete para una entrada de usuario en un diagrama UID`

El segundo paso es definir un test que, ademas de verificar los aspectos de interfaz y navegación, nos permita corroborar el funcionamiento de la característica RIA incorporada a la entrada de datos (en este caso el *autocomplete*).

Utilizaremos el *plugin* de Selenium: *Selenium IDE*. Este *plugin*, para el navegador *Mozilla Firefox*, permite definir (manualmente o grabando acciones) y ejecutar tests de interacción, así como también traducir el código generado a las APIs de diferentes lenguajes de programación como Java y Python.

Para incorporar esta funcionalidad en los *UI Mockups* utilizaremos el *YUI Autocomplete Component* provisto por la librería YUI. Nuestro *UI mockup* tendrá una simulación de esta funcionalidad con algunos valores locales predefinidos para las sugerencias, que luego se reemplazaran por una solicitud remota cuando derivemos los prototipos de nuestra aplicación. La figura 7.4 muestra un *UI Mockup* definido para nuestro UID, en cual agregamos funcionalidad a la entrada de texto mediante *YUI Autocomplete Component*, junto con las utilidades: *Firebug* (para obtener las expresiones *XPath*), y Selenium IDE (para definir el test de navegación e interacción).

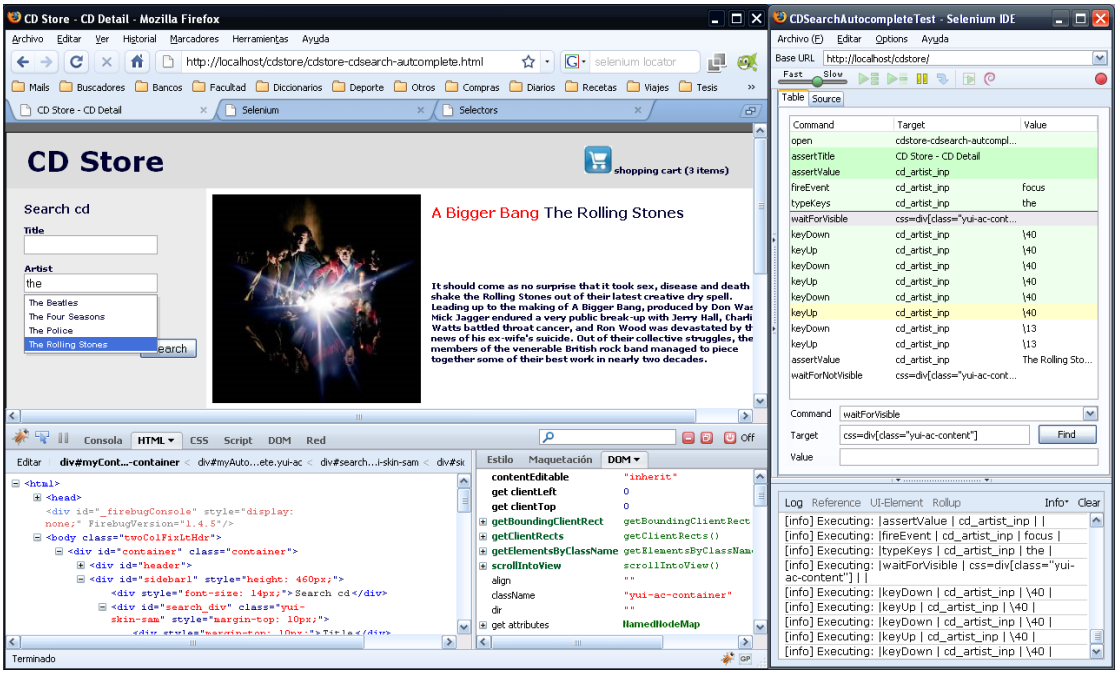

Figura 7.4: `UI Mockup ampliado con YUI Autocomplete Component y las utilidades Selenium IDE y Firebug`

Los pasos para un posible test que verifique el autocompletado serían:

1. Tipear texto parcialmente en la entrada de datos
2. Esperar por visibilidad del pop-up con las sugerencias
3. Seleccionar un ítem en particular
4. *Assert*: el texto de la entrada es igual al texto del ítem seleccionado
5. *Assert*: el pop-up no es visible luego de seleccionar un ítem

Primero se simula el tipeo parcial de cierta cantidad de caracteres (los primeros caracteres de un nombre de compositor, por ejemplo *The Roll* en alución a *The Rolling Stones*); se espera hasta que la lista desplegable sea visible realizando una aserción sobre esta condición; se selecciona un ítem de la lista desplegable; y por ultimo se realizan aserciones sobre el valor actual de la entrada de texto (cuyo valor actual debería ser el mismo que el ítem previamente seleccionado) y que la lista de sugerencias no este visible (luego de realizar la selección del ítem). La figura 7.5 muestra la versión de nuestro test en acciones disponibles en *Selenium IDE*.

```
 assertTitle: "CD Store - CD Detail"
#la entrada esta vácia antes de seleccionar un ítem
 assertValue: cd_artist_inp == ""
 fireEvent: cd_artist_inp focus
#tipeo parcial de texto
 typeKeys: cd_artist_inp "the"
#pop-up visible
 waitForVisible: css=div[class="yui-ac-content"]
#se simula la selección del cuarto ítem
 keyDown: cd_artist_inp \40
 keyUp: cd_artist_inp \40
 keyDown: cd_artist_inp \40
 keyUp: cd_artist_inp \40
 keyDown: cd_artist_inp \40
 keyUp: cd_artist_inp \40
 keyDown: cd_artist_inp \40
 keyUp: cd_artist_inp \40
#se presiona enter para seleccionar el ítem actual
 keyDown: cd_artist_inp \13
 keyUp: cd_artist_inp \13
#la entrada ahora tiene como valor el ítem seleccionado
 assertValue: cd_artist_inp == "The Rolling Stones"
#pop-up no visible
 waitForNotVisible: css=div[class="yui-ac-content"]
```

Figura 7.5: Código generado por Selenium IDE para verificar el autocompletado, luego puede ser traducido a una API de alguno de los lenguajes de programación soportados por el framework

Como pre-condición para la ejecución de este test necesitamos que el valor ingresado, en la entrada de datos, genere al menos un ítem sugerido que luego podamos seleccionar. Cuando ejecutemos el test sobre un *UI mockup* podemos tener algunos valores predefinidos para la lista de sugerencias; y luego, en los sucesivos prototipos generados de la aplicación, obtener los valores para las sugerencias directamente de una base de datos de la aplicación final.

Este fragmento de código puede ser inyectado en el código del test de unidad de navegación e interacción y así ampliar la cantidad de componentes y aspectos de navegación verificados. Es importante resaltar que este test es solo una de muchas variantes para verificar el comportamiento correcto del patrón RIA *autocomplete*. Podemos definir otros test que verifiquen mas a fondo la característica de auto-completado. Por ejemplo, podemos verificar que ciertos ítems sugeridos desaparecen de la lista al ingresar mas caracteres, o ingresar caracteres que fuercen a que no haya ningún ítem para sugerir con lo cual no debería mostrarse la lista de sugerencias, etc.

7.4. Inline validation

La validación en línea (*inline validation*) permite verificar que los datos ingresados por el usuario cumplan con ciertas propiedades en el mismo momento en que son ingresados. En general, se puede ver esta característica en las entradas de texto estándar de un formulario de una aplicación Web. Por ejemplo, validar si una dirección de correo electrónico tiene el formato correcto o la generación de una clave de usuario tiene determinada longitud mínima requerida. Esto permite eliminar pasos intermedios de validación con el servidor, ya que se asegura que la información enviada es correcta antes de pasar al siguiente estado de interacción.

7.4.1. Descripción del patrón

Descripción del problema

- Encontrar y corregir errores en los datos ingresados es una tarea adicional que no permite completar la tarea primaria iniciada por el usuario.

Solución

- Validar las entradas de usuario tan pronto como sea posible y mostrar mensajes de validación en línea (*feedback*) sobre la entrada de datos que esta siendo objeto de la validación.

Justificación

- Como regla general, se mejora la interacción cuanto mas rápido se pueda validar la entrada de usuario y proveer *feedback* relacionado. Esto se debe a que el usuario esta concentrado en el objeto que esta siendo validado (o acaba de terminar con el), por lo tanto el contexto sigue siendo inmediato y sigue presente en su mente. Mostrando los mensajes en línea se refuerza el contexto evitando que el usuario tenga que crear un linea mental entre el mensaje de validación y el problema en la entrada de datos.

Contexto

- Se necesita dejar espacio para ubicar los mensajes de validación en línea e indicadores visuales.
- Se puede realizar la validación de manera incremental, a medida que el usuario provee una entrada sobre la cual validar.
- Se debe validar dentro de un marco de tiempo relativamente corto (relativo a la entrada y cuan rápido se desplaza el usuario).

Implementación

- De ser posible (técnicamente hablando), ejecutar la validación inmediatamente después que la entrada de datos pierde el foco. En algunos casos también tiene sentido la validación inmediata (como por ejemplo cuando se produce un evento de presionar una tecla o seleccionar una opción), pero teniendo en mente que no se debe bloquear al usuario en un bucle de entrada que pueda resultar frustrante.
- Realizar la validación de manera asincrónica si esta toma mas de algunos pocos milisegundos. En el caso que la validación requiera un tiempo considerable, tal vez sea necesario repensar el enfoque. Si la validación no es casi inmediata, el efecto que causara mostrar un mensaje de validación será mas sorpresivo y confuso que si el mismo ocurre luego de que el usuario de alguna indicación de que ha finalizado (por ejemplo, clickear un botón de submit). Si la validación requiere de aproximadamente un segundo, se debe considerar mostrar algún tipo de mensaje que indique que se esta procesando la entrada, para que el usuario sepa lo que esta sucediendo; de esta manera el usuario esperará por un resultado y realizará una pausa manteniendo el foco de atención en la entrada de datos en cuestión.
- Mostrar los mensajes de problemas en la validación en línea junto con el objeto que esta siendo validado. Es preferible mostrar un mensaje (y no solo algún tipo de marca como un asterisco o signo de exclamación) con el problema si hay espacio disponible para ubicarlos. En el caso que tengan que estar separados, se necesita mostrar el mensaje en una posición consistente que este en la vista del usuario y también muestre un indicador al lado del objeto de datos cuya validación fallo. Es muy frustrante para los usuarios tratar de terminar un tarea y estar bloqueado sin poder ver o entender la razón de la falla.

Ejemplos

Como primer ejemplo podemos citar a la página de registración de nuevos usuario de *Yahoo Mail!* [YMail09] (figura 7.6). En este caso, se hace uso de varias de las sugerencias descriptas en la implementación del patrón.

Figura 7.6: `Parte del formulario de registración de Yahoo Mail! en el cual se observa la utilidad del patrón de diseño inline validation`

- Al situar al foco sobre uno de los campos de entrada, se muestra *feedback* en línea acerca de la información que se debe completar.
- Se validan los campos cuando los mismos pierden el foco salvo para el caso de la selección del nombre de usuario. En este ultimo caso, el usuario debe oprimir el botón comprobar para iniciar la validación. Esta implementación tiene sentido ya que validar la existencia de la cuenta de usuario demanda algunos segundo de procesamiento y no sería adecuado ejecutar este proceso de validación en cada carácter ingresado por el usuario.
- En el caso de la definición de la clave de usuario, la validación se realiza inmediatamente a medida que el usuario agrega caracteres a su clave. Hay un indicador que se actualiza con cada carácter tipeado, marcando el nivel de seguridad alcanzado por la clave ingresada.
- En todos los casos se muestran los mensaje de las validaciones fallidas junto al campo requerido. Esto facilita el reconocimiento de cual es el error y permite al usuario corregirlo rápidamente.

Este es un buen ejemplo de utilización del patrón *inline validation*. En contraste, en una aplicación Web tradicional el usuario deberá enviar una y otra vez el formulario al servidor para lograr

corregir todos los errores. Con la validación en línea se puede asegurar que una vez enviado el formulario todos los datos ya han sido validados.

Otro ejemplo de validación en línea se observa en el formulario de registración del sitio *Codeplex* [Codepl09]. La figura 7.7 muestra el formulario de registración donde se puede observar que se válida el formato del texto ingresado en el campo de dirección de correo electrónico, para determinar si el mismo tiene justamente el formato de una dirección de correo electrónico válida.

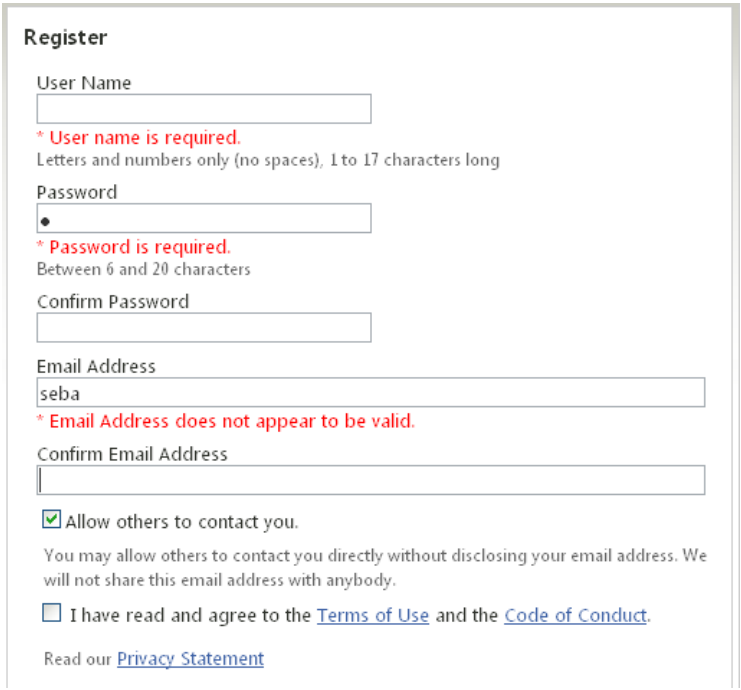

Figura 7.7: `Formulario de registración del sitio Codeplex, donde se utiliza el patrón de diseño de validación en línea`

7.4.2. Definición de test de navegación e interacción

El patrón de validación en línea se puede aplicar a las entradas de usuario. Tomemos como ejemplo el diagrama UID, mostrado en la figura 7.8, en el cual deseamos especificar que la entrada de usuario del *e-mail* posee la propiedad *inline-validation*.

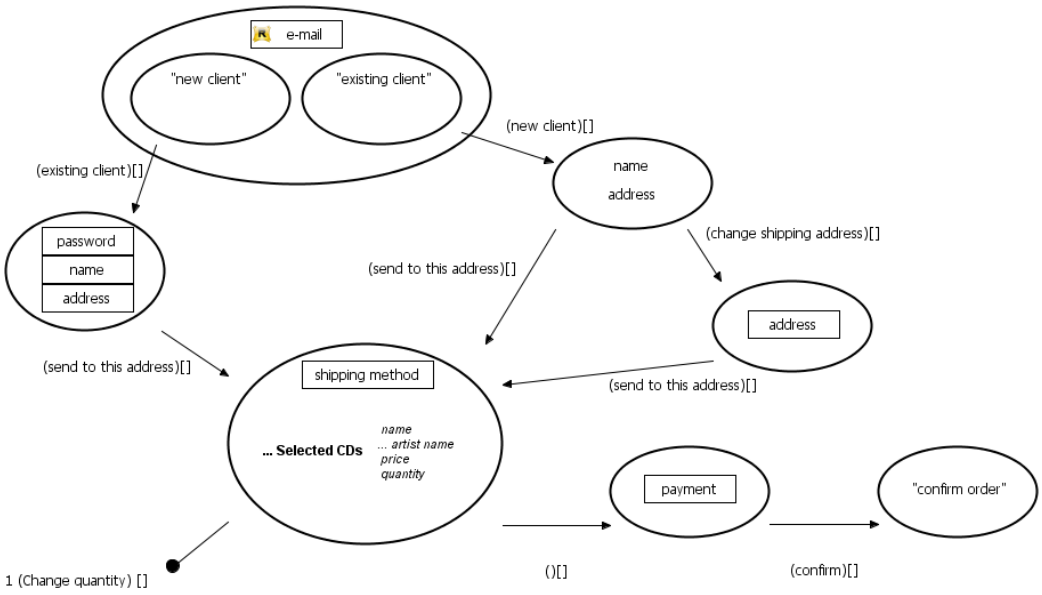

Figura 7.8: `Diagrama UID: checkout. La entrada de usuario: e-mail, tiene activado el patrón inline-validation, con lo cual se validará el formato correcto de la entrada antes de pasar a la siguiente interacción`

Ahora pasemos a la definición de los test que permitirán verificar que la propiedad RIA funciona correctamente en la entrada de usuario. Al igual que en el patrón *autocomplete* podríamos definir varios test para verificar el funcionamiento de la propiedad en diferentes situaciones. En primer lugar simularemos el ingreso de una entrada válida. Los pasos para definir esta situación serían:

1. Ingresar una dirección de mail sintácticamente correcta.
2. Simular el cambio de foco de control.
3. *Assert*: el mensaje de validación no es visible.
4. Ingresar el resto de la información requerida
5. Simular el envío del formulario
6. *Assert*: cambio la ubicación actual (se paso a una nueva pagina o contexto de interacción)

Ingresamos una dirección de correo electrónico sintácticamente correcta; luego verificamos que no hay un mensaje de error visible; por último completamos el resto de la información y realizamos el *submit* del formulario. Como no debería existir error en la entrada de usuario entonces verificamos que estamos en un nuevo estado de interacción.

Tomando como ejemplo el formulario del sitio *Codeplex* (figura 7.7) podemos definir el test para la situación planteada utilizando código *Selenium* (figura 7.9).

```
 assertTitle: "CodePlex - Registration"
 type: UserName "user"
 type: Password "******"
 type: PasswordConfirmation "******"
#Verificamos que no esta visible el mensaje de error
 assertElementNotPresent: //*[@id="MvcValidation_EmailAddressErrorMessage"]
#Tipeamos una dirección de mail válida
 type: EmailAddress "user@user.com.ar"
 type: EmailAddressConfirmation "user@user.com.ar"
 click ReadTerms
#Verificamos que no se mostró el mensaje de validación, ya que la información ingresada es válida
 assertElementNotPresent: //*[@id="MvcValidation_EmailAddressErrorMessage"]
#Enviamos el formulario
 click submitButton
#Como la información ingresada es válida, deberíamos haber cambiado de contexto
 assertNotLocation: */site/register
```

Figura 7.9: `Definición en Selenium de test para verificar el patrón Inline Validation, en el caso de una entrada válida`

La segunda situación de intereses es verificar el comportamiento de la aplicación cuando se simula el ingreso de una entrada inválida. Los pasos para un test que describa esta situación serían:

1. Ingresar una dirección de mail sintácticamente incorrecta.
2. Simular el cambio de foco de control.
3. *Assert*: el mensaje de validación es visible.
4. Ingresar el resto de la información requerida
5. Simular el envío del formulario
6. *Assert*: no cambiamos el estado de interacción actual

Se ingresa una dirección de correo electrónico incorrecta; luego se verifica que el mensaje de error en la validación este visible; por ultimo al intentar *submitir* el formulario eso no debería ser posible hasta que no se corrija el error en la entrada de usuario, por lo cual verificamos que seguimos en el mismo estado de interacción. La figura 7.10 muestra el código Selenium para esta situación para el formulario de registración del sitio *Codeplex*.

```
 assertTitle: CodePlex - Registration
 type: UserName: "user"
 type: Password: "******"
 type: PasswordConfirmation: "******"
#Verificamos que no esta visible el mensaje de error
 assertElementNotPresent: //*[@id="MvcValidation_EmailAddressErrorMessage"]
#Tipeamos una dirección de mail inválida
 type: EmailAddress "user"
 type: EmailAddressConfirmation "user"
 click: ReadTerms
 focus: submitButton
#Verificamos que se mostró el mensaje de validación, ya que la información ingresada es inválida
 assertVisible: //*[@id="MvcValidation_EmailAddressErrorMessage"]
#Enviamos el formulario
 click: submitButton
#Seguimos en la misma ubicación
 assertLocation: */site/register
```

Figura 7.10: `Definición en Selenium de test para verificar el patrón Inline Validation, en el caso de una entrada inválida`

En el caso de tener mensajes de *feedback* en línea, como en el ejemplo de *Yahoo Mail!*, se puede agregar en los test la verificación de que dichos mensajes son visibles cuando una entrada obtiene el foco.

Esta funcionalidad la podemos agregar a nuestros *UI Mockups* mediante el uso de JavaScript y estilos, suponiendo una aplicación realizada con HTML. Básicamente, podemos verificar los datos ingresados cuando las entradas de usuario pierden el foco y mostrar los mensajes de validación cambiando la propiedad visible, de las hojas de estilo, de los contenedores que muestran tales mensajes.

7.5. Mouse hover

Las aplicaciones RIA proveen una interacción mucho mas rica y compleja entre los usuarios y la interfaz de la aplicación. Entre estas interacciones se encuentran aquellas generadas al posicionar el cursor del *mouse* u otro dispositivo de señalamiento sobre los diferentes objetos que componen la interfaz. En esta sección describiremos dos patrones relacionados con este tipo de interacciones: *hover tooltip* y *hover detail*.

7.5.1. Hover Tooltip

El patrón *Hover Tooltip* o *Tooltip Invitation* nos permite indicar, mediante un mensaje breve, que es lo que podemos hacer con un objecto *clickeable*. Esta descripción será mostrada al pasar el cursor sobre el objeto con el cual interactuar. Podemos tener variantes donde el usuario explicitamente deba cerrar el *tooltip* una vez mostrado. La figura 7.11 muestra diferentes estilos para implementar *tool tips*.

7.5.1.1. Descripción del patrón

Alias

- Tooltip Invitation

Descripción del problema

Figura 7.11: `Diferentes alternativas para mostrar tooltips sobre objetos de interacción`

- Los diseñadores desean llamar la atención del usuario acerca de lo que sucederá si clickean el objeto debajo del cursor del mouse.

Cuando utilizarlo

- Se desea invitar o atraer al usuario a clickear o interactuar con el objeto debajo del cursor.
- Se desea dejar en claro que se producirá un evento cuando el usuario clickee el objeto debajo del cursor.
- El usuario interactúa directamente con el objeto (por ejemplo, en la edición en línea).
- Se necesita una descripción textual para dejar explicito que ocurrirá un evento cuando el usuario interactúe con el objeto.

Justificación

- Las invitaciones son atractivas cuando se desea:
  - que el usuario descubra una nueva característica;
  - introducir al usuario a una nueva forma de interactuar (nuevo idioma);
  - que la interacción sea mas fluida.

Solución

- Proveer un *tooltip* cuando el mouse pasa sobre el área interactiva del objeto.
- Mostrar el *tooltip* dentro de un periodo corto de tiempo o inmediatamente cuando el mouse se posiciona sobre el área de interacción.
- Mantener el *tooltip* visible mientras el mouse permanezca sobre el objeto de interacción.
- Ocultar el *tooltip* cuando el mouse deje el área de interacción.
- Proveer una frase corta y descriptiva (en general verbos) que llamen o inviten al usuario a clickear sobre los objetos.

Ejemplos

En la figura 7.12 se observa el sitio *Flickr* [YFlickr09] de Yahoo! que utiliza intensivamente el patrón *hover tooltip*, para describir las diferentes operaciones que se pueden realizar sobre una fotografía.

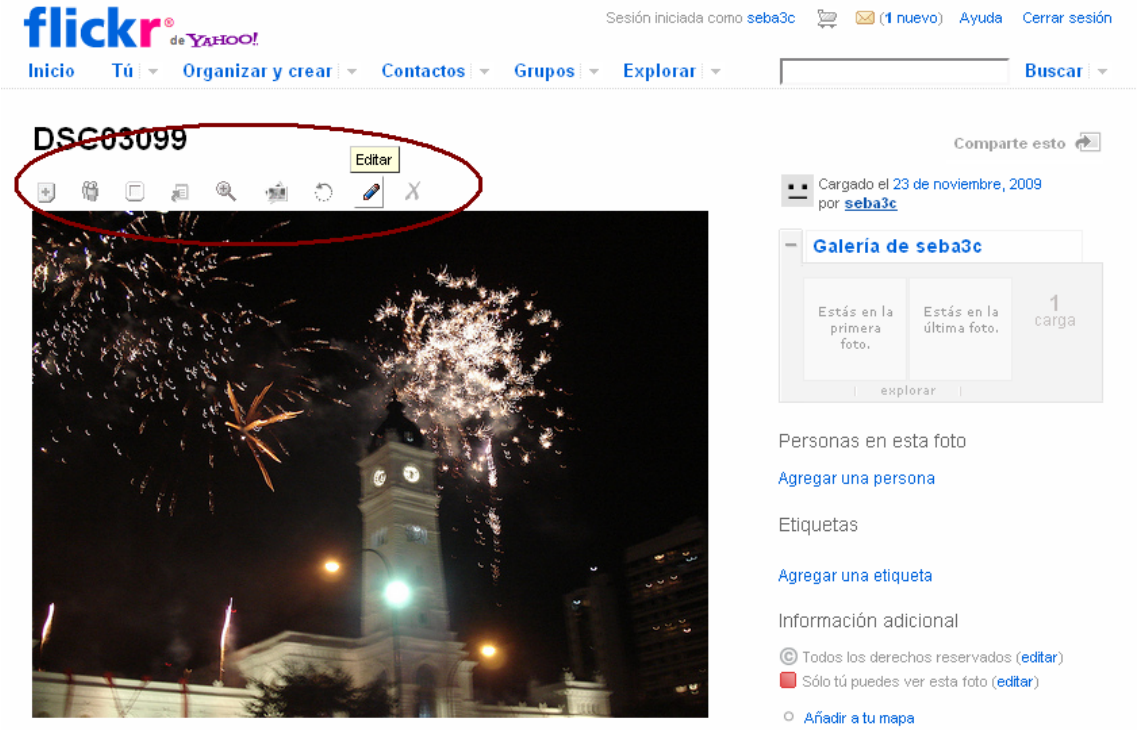

Figura 7.12: `Utilización del patrón de diseño Hover tool tip en el sitio de Yahoo! Flickr`

7.5.1.2. Definición de test de navegación e interacción

Tomemos como ejemplo el diagrama UID de la figura 4.2. Queremos especificar que al clickear sobre la imagen de la tapa de un CD podemos ver la misma en un tamaño mayor (se realiza un zoom de la imagen). Es decir, deseamos especificar un *tooltip* que diga algo similar a *Click to enlarge* (Clickear para agrandar) cuando el mouse se posiciones sobre el ítem de datos *album cover* que conforma la estructura de un CD.

Definir un test para esta característica resulta bastante sencillo. Una buena forma de testear que el *tooltip* especificado funciona correctamente podría ser:

1. Posicionar el mouse sobre el objeto de interacción.
2. *Assert*: el tool tip es visible.
3. *Assert*: el texto del tool tip es el especificado en los requerimientos
4. Posicionar el mouse fuera del objeto de interacción.
5. *Assert*: el tool tip no es visible.

En primer lugar, posicionamos el mouse sobre el objeto para el cual definimos el *tooltip*; verificamos que el *tooltip* esta visible y su texto es el especificado por los requerimientos; posicionamos el mouse fuera del área de interacción; y por ultimo verificamos que el *tooltip* ahora esta oculto.

Para implementar un *tooltip* disponemos del componente *Tooltip Control* provisto por la librería YUI. La imagen 7.13 muestra nuestro *UI Mockup* con la implementación de un *tool tip*. La figura 7.14 muestra la definición en Selenium de nuestro test para verificar la funcionalidad RIA.

Figura 7.13: `UI Mockup con implementación de hover tooltip mediante el componente YUI Tooltip Control`

```
#Posicionamos el mouse sobre la imagen
 mouseOver //*[@id="cd_cover_img"]
#Esperamos a que el tooltip este visible
 waitForVisible //*[@id="imgTooltip"]
#Verificamos el contenido del tooltip
 assertText //*[@id="imgTooltip"] Click to enlarge.
#Posicionamos el mouse fuera de la imagen
 mouseOut //*[@id="cd_cover_img"]
#Esperamos a que el tooltip no este visible
 waitForNotVisible //*[@id="imgTooltip"]
```

Figura 7.14: `Definición de test en Selenium para verificar el funcionamiento de hover tooltip`

7.5.2. Hover detail

El patrón *hover detail* es otra variante de eventos RIA generados cuando el cursor del mouse se posiciona sobre un objeto particular. A diferencia del patrón *hover tooltip* donde simplemente mostramos un pequeño texto descriptivo, en el *hover detail* el objetivo es ampliar la información de una determinada entidad mediante un *popup* en línea (manteniendo el contexto de la página actual). Por ejemplo, mostrar una breve descripción de un CD, cuando nos posicionamos sobre el ítem título.

7.5.2.1. Descripción del patrón

Descripción del problema

- Evitar cambiar el contexto para mostrar información adicional.

Cuando utilizarlo

- Se desea mostrar detalles o información adicional en contexto.

Solución

- Mostrar un *popup* en contexto con la información adicional cuando el mouse esta sobre el área interactiva del objeto.
- Mostrar el *popup* inmediatamente o dentro de un periodo corto de tiempo cuando el mouse se posiciona sobre el área de interacción.
- Mantener el *popup* visible mientras el mouse permanezca sobre el objeto de interacción.
- Ocultar el *popup* cuando el mouse deje el área de interacción o se realice un click.

Ejemplos

Dos ejemplos de la utilización de este patrón se pueden observar en el sitio de renta de DVDs netflix [Netfli09] (figura 7.15) y en Yahoo! News [YNews09] (figura 7.16).

Figura 7.15: `Hover detail en netflix`

Figura 7.16: `Hover detail en Yahoo! News`

7.5.2.2. Definición de test de navegación e interacción

La característica RIA *hover detail* la podemos asociar a los ítems de datos (*data items*) de los diagramas UIDs, seteando una estructura (*Structure*) a la propiedad *hoverDetail* que representará el detalle de información asociado al ítem de datos en cuestión.

Supongamos que queremos especificar que los nombres de los compositores tienen asociado un detalle con una breve biografía en el diagrama UID de la figura 4.3. A tal fin setearemos la propiedad *hoverDetail* del ítem de datos *artist name* con una estructura que representará la biografía resumida. Los pasos para definir un test que verifique esta característica son muy similares a los especificados para *hover tooltip* con la diferencia que podemos realizar mas aserciones acerca del contenido interno del pop-up:

1. Posicionar el mouse sobre el objeto de interacción.
2. *Assert*: el pop-up es visible.
3. *Assert*: la contenido interno del pop-up es el especificado en los requerimientos
4. Posicionar el mouse fuera del objeto de interacción.
5. *Assert*: el pop-up no es visible.

Podemos implementar esta característica mediante el *YUI Panel Component* de la librería YUI. La figura 7.17 nos muestra un *UI Mockup* que hace uso de *hover detail* para mostrar una pequeña biografía del artista de cada CD. La figura 7.18 define un test para este *UI Mockup* donde verificamos el comportamiento básico de esta característica comprobando que el *pop-up* se oculta tanto moviendo el mouse fuera del área de interacción como realizando un click.

Figura 7.17: `UI Mockup con implementación de hover detail mediante el componente YUI Panel`

```
#Verificamos que el pop-up no es visible
 verifyNotVisible: //*[@id="artist_bio_1"]
#Posicionamos el mouse sobre nombre del artista del CD
 mouseOver: //*[@id="artistCD1"]
#Esperamos que el pop-up sea visible
 waitForVisible: //*[@id="artist_bio_1"]
#Posicionamos el mouse fuera del área de interacción
 mouseOut: //*[@id="artistCD1"]
#Esperamos que el pop-up no este visible
 waitForNotVisible: //*[@id="artist_bio_1"]
#Repetimos los pasos pero ahora realizamos un click para ocultar el pop-up
 mouseOver: //*[@id="artistCD1"]
 waitForVisible: //*[@id="artist_bio_1"]
 click: //*[@id="artistCD1"]
 waitForNotVisible: //*[@id="artist_bio_1"]
```

Figura 7.18: `Definición de test en Selenium para verificar el funcionamiento de hover detail`

Una diferencia importante entre el contenido de un *pop-up* y un simple *tooltip* es que el primero ademas de tener un contenido mas complejo podría potencialmente contener enlaces a otras páginas. Tal es el caso del correo *Gmail* [Gmail09], en donde al posicionarse sobre la dirección de correo electrónico de un contacto se muestra un *pop-up* con información adicional del contacto y enlaces con operaciones tales como *enviar un email*, *chatear*, o *ver el historial de conversaciones* (figura 7.19).

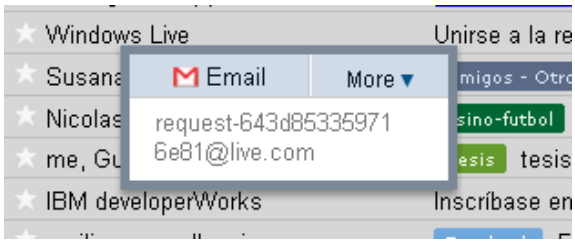

Figura 7.19: `Hover detail en Gmail, ademas de información adicional el pop-up presenta enlaces que pueden cambiar el contexto actual`

En un caso como este, podríamos desear verificar la navegabilidad o cambio de contexto al utilizar las operaciones que muestra el *pop-up*. Las operaciones las podemos especificar como parte de la estructura que define nuestro *pop-up*. Es decir, serian ítems de datos adicionales contenidos en el pop-up. Para verificar la navegabilidad podríamos definir un test que consista de los siguiente pasos:

1. Posicionar el mouse sobre el objeto de interacción.
2. *Assert*: el pop-up es visible.
3. *Assert*: la contenido interno del pop-up es el especificado en los requerimientos
4. Clickear una operación contenida en el pop-up
5. *Assert*: cambio el contexto a la página a la cual nos redirecciona la operación seleccionada

Los pasos 1 a 3 son exactamente los mismos que los especificados para verificar la funcionalidad de *hover detail*. De esta manera, con estos dos test podríamos por un lado verificar el funcionamiento del *pop-up* y por otro lado la navegabilidad que surge de las operaciones adicionales provistas.

7.6. Deferred content loading

El patrón de diseño *deferred content loading* (carga de contenido diferida), también llamado *lazy loading*, nos permite especificar que cierto contenido del sitio Web pueda ser cargado en la estructura de la página en forma diferida al resto del contenido. En general, se aplica a contenido adicional de gran tamaño o que requiere un tiempo considerable en estar disponible al usuario como pueden ser imágenes y vídeos.

7.6.1. Descripción del patrón

Alias

- Lazy loading

Problema que resuelve

- Optimiza el rendimiento de carga de un sitio Web.

Cuando utilizarlo

- El sitio posee contenido periférico o adicional cuya carga puede ser lenta en comparación al resto de la información.

Solución

- Se debe utilizar con moderación, en general para mejorar el rendimiento de carga de la página principal de un sitio Web.
- Utilizar un indicador de carga, que permita al usuario notar que en ese lugar se mostrará un contenido que aun no esta disponible.
- Es útil para información externa al sitio Web. Por ejemplo, información obtenida desde un *Web Service* en otro *Host*.

Ejemplos

Como ejemplo del uso de este patrón podemos volver a citar el sitio Web de *Wilson* [Wilson09] (figura 3.5). Cada vez que se selecciona un modelo de raqueta, para ver su especificación, tenemos casi instantáneamente la descripción del modelo, mientras que la imagen ampliada de la raqueta toma un par de segundos en cargarse mostrando un indicador de carga durante este proceso.

7.6.2. Definición de test de navegación e interacción

Definir un test para este patrón es muy simple. En primer lugar, debemos especificar aquellos ítems de datos de nuestro diagrama UID que presentan esta característica. Por ejemplo, para el diagrama UID de la figura 4.2, podemos especificar que el ítem de datos *album cover* de la estructura CD tiene carga diferida, seteando la propiedad *deferredLoading* en `True`.

La definición de un test para esta característica, se puede reducir a tan solo cargar la URL de la página y esperar a que dicho elemento este presente dentro de la estructura del documento. En el caso de *Selenenium* tenemos disponibles la familia de métodos *waitForX()*, justamente pensados para aplicaciones AJAX donde parte del contenido de una página puede cargarse asincrónicamente y no estar disponible en un determinado instante. Por su parte, la librería YUI nos provee de la utilidad *Image Loader* para manejar la carga diferida de las imágenes mostradas en nuestra

aplicación. En la imagen 7.20 se muestra el *UI mockup* correspondiente a nuestro listado de CDs donde las imágenes de los álbumes se cargan en forma diferida al resto de la estructura de la página; mientras que en la figura 7.21 se muestra el código Selenium para verificar el funcionamiento de la carga diferida de las imágenes, en este caso los pasos del test son:

1. Verificar que el atributo *src* de los *tags IMG* no estén presente.
2. Esperar a que el atributo *src* de cada imagen sea seteado.
3. Verificar que el atributo *src* esta presente en cada *tag IMG* cargado en forma diferida.

Figura 7.20: `UI Mockup con implementación de deferred content loading mediante el componente YUI Image Loader`

```
#Verificamos que el atributo src no existe en las imágenes
 verifyElementNotPresent: //*[@id="cdcover1"]/@src
 verifyElementNotPresent: //*[@id="cdcover2"]/@src
 verifyElementNotPresent: //*[@id="cdcover3"]/@src
#esperamos a que se setee el atributo src de cada tag IMG
 waitForAttribute: //*[@id="cdcover1"]/@src imgs/itsonlyrockandroll-small.jpg
 waitForAttribute: //*[@id="cdcover2"]/@src imgs/abiggerbang-small.jpg
 waitForAttribute: //*[@id="cdcover3"]/@src imgs/vodoolounge-small.jpg
#Verificamos que el atributo src esta presente en cada imagen
 verifyElementPresent: //*[@id="cdcover1"]/@src
 verifyElementPresent: //*[@id="cdcover2"]/@src
 verifyElementPresent: //*[@id="cdcover3"]/@src
```

Figura 7.21: `Definición de test en Selenium para verificar el funcionamiento de deferred content loading`

7.7. Resumen

En este capitulo nos centramos en como ampliar nuestros test de unidad de navegación para que ademas de seguir verificando aspectos de interfaz y navegación, sean capaces de verificar el funcionamiento de algunos componentes RIA. Si bien utilizamos para los ejemplos aplicaciones híbridas compuestas por páginas estándar HTML y componentes RIA, y el *framework Selenium* para la definición de los test; las ideas descriptas pueden ser utilizadas en el testing de aplicaciones desarrolladas con otras tecnologías y mediante la utilización de un *framework* de testing Web que soporte la interacción con las interfaces de la tecnología en cuestión.

Capítulo 8

Conclusiones, críticas y trabajos futuros

8.1. Conclusiones

Cada vez son mas los sitios y portales Web desarrollados integramente o que incluyen componentes de tecnologías RIA. El desarrollo de estas aplicaciones presenta un grado de complejidad mayor a las tradicionales aplicaciones Web, con lo cual son mas propensas a contener errores sino se las desarrolla con una metodología adecuada y no se incorpora el testing de todos sus componentes como una tarea primaria durante todo el desarrollo. Por este motivo, la propuesta de este trabajo es desarrollar aplicaciones Web y aplicaciones RIA mediante la utilización de una metodología dirigida por modelos, que permiten obtener los componentes fundamentales de una aplicación mediante la derivación libre de errores de modelos, junto con la inclusión de aspectos de la metodología ágil del desarrollo dirigido por pruebas TDD aplicados no solo a los componentes de lógica de negocios (test de unidad tradicionales), sino también a los aspectos de interfaz y navegación (test de unidad de navegación) propios de cualquier aplicación Web. De esta manera, la utilización de esta metodología en el desarrollo Web tiene por objetivo asegurar que cada nuevo requerimiento y modificación/refactoring realizados en la aplicación funciona correctamente y no corrompe la funcionalidad agregada y testeada previamente.

Haciendo énfasis en las actividades de testing, a diferencia de lo que sucedía en décadas anteriores el testing ha pasado a formar parte fundamental de cualquier desarrollo de software que pretenda asegurar un mínimo de calidad en prestaciones y funcionamiento. Hoy es imposible pensar en realizar un desarrollo de cualquier sistema de software (aplicación de escritorio, aplicación Web, sistema de tiempo real, etc.) que no incluya al menos un mínimo de actividades de testing que permitan asegurar la calidad del desarrollo. En mi experiencia personal he sido parte de un área de QA (*Quality Assurance*) y he aplicado el concepto de TDD en mis desarrollos obteniendo muy buenos resultados, que tal vez desarrollando de manera mas tradicional (solo enfocándose en la implementación y quizás luego definiendo los test de unidad) hubiese consumido numerosas horas en intentar detectar los errores mediante revisión de código y *debugging*.

Con respecto a los temas investigados en este trabajo, he podido incursionar en muchos conceptos y áreas que hasta el momento eran desconocidas o apenas había escuchado nombrar en alguna ocasión. En particular, los conceptos de aplicaciones RIA, las metodologías MDWE y los frameworks para testing de navegación e interacción Web.

8.2. Críticas y mejoras posibles

Con respecto a la metodología propuesta, se puede discernir en el hecho de que su aplicación requiere de un equipo muy organizado y riguroso a la hora de realizar cada una de las actividades descriptas, y que muchos podrían argumentar que la inclusión de todas las herramientas y pasos de desarrollo propuestos consumirían demasiado tiempo y esfuerzo. Sin embargo, a su favor podemos refutar esto diciendo que una vez que se logren aceitar todas las actividades y se haga un uso correcto de todas las herramientas se puede obtener un desarrollo incremental libre de errores y con la seguridad que al alcanzar el desarrollo final este cumplirá con los requerimientos propuestos en un ciento por ciento.

Refiriéndonos a los desarrollos realizado en este trabajo: el editor de diagramas UID implementado es solo una propuesta inicial (a modo ilustrativo), ya que su funcionalidad es muy básica y limitada. Solo incluye la posibilidad de describir algunos comportamientos RIA y no posee ningún tipo de validaciones, como por ejemplo verificar que el diagrama este bien formado. Tampoco brinda el soporte necesario para la derivación de los *templates* de los test de navegación.

Otro punto criticable es el hecho de si incluir los requerimientos de comportamiento RIA en los diagramas UIDs es la mejor opción. Como vimos en el trabajo existen otras alternativas a esto que podrían evaluarse o incluirse como parte del proceso de desarrollo.

Todos los temas vistos durante el trabajo tienen como objetivo proponer una alternativa mas para mejorar el desarrollo de las aplicaciones Web.

8.3. Trabajos Futuros

Es mas que evidente que todos los temas tratados en este trabajo son tan solo la punta del iceberg y muchas de las actividades del proceso de desarrollo descriptas requieren mejoras y/o una investigación por separado. Entre las líneas de investigación que se pueden desprender a partir de este trabajo podemos citar las siguientes:

- `MDWE y modelado de aplicaciones RIA`: proponer y/o ampliar las metodologías MDWE existentes para soportar el modelado y derivación de aplicaciones RIA.
- `Frameworks para testing Web`: investigar los frameworks para testing Web disponibles para cada una de las tecnologías RIA del mercado. También se podría realizar una ampliación de alguno de los *frameworks* existentes para soportar otras tecnologías RIA (tal es el caso de las ampliaciones existentes para Selenium) o proponer la implementación de un nuevo framework orientado a alguna tecnología particular.
- `Derivación automática de test`: derivar los test de navegación en forma automática a partir de los diagramas UIDs o utilizando alguna otra herramienta adecuada para la captura y especificación de requerimientos.
- `Diseño de UI Mockups`: investigar herramientas que permitan la fácil y rápida implementación y diseño de los *UI Mockups*, útiles en la captura de requerimientos y definición de los test de navegación.
- `Patrones de diseño RIA`: investigar los diferentes patrones RIA y como definir test de navegación e interacción que permitan verificar su correcto funcionamiento dentro de la aplicación Web desarrollada.

Bibliografía

[Adobe09] *Adobe Systems* [en línea] `http://www.adobe.com` [consultado : 29/11/2009]

[AjaxP09] *AjaxPatterns* [en línea] `http://ajaxpatterns.org` [consultado : 29/11/2009]

[Allai02] J. Allaire. *Macromedia Flash MX-A next-generation rich client.* Technical report, Macromedia, Marzo de 2002 [en línea] `http://www.adobe.com/devnet/flash/whitepapers/richclient.pdf` [consultado : 13/12/2009]

[Amp09] S. W. Ampler. *Introduction to Test Driven Design.* Agile Data, 2009 [en línea] `http://www.agiledata.org/essays/tdd.html` [consultado : 27/11/2009]

[Amazon09] *Amazon* [en línea] `http://www.amazon.com` [consultado : 29/11/2009]

[AOLMT09] AOL, *Music Top 100 videos* [en línea] `http://music.aol.com/help/syndication/desktop-widgets` [consultado : 29/11/2009]

[Ast03] D. Astels. *Test-Driven Development: A Practical Guide.* New Jersey: Prentice Hall, 3ra Edición , c2003. [en línea] `http://cid-dba5855b5919078d.skydrive.live.com/self.aspx/Public/Software/Engineering/Testing/test-driven-development-a-practical-guide.chm` [consultado : 27/11/2009]

[BCFT06a] A. Bozzon, S. Comai, P. Fraternali, G. Toffetti Carughi. *Conceptual Modeling and Code Generation for Rich Internet Applications.* En: Proceedings of the International Conference on Web Engineering ICWE2006 (6, Palo Alto [California, USA], 11-14 de Julio de 2006).

[BCFT06b] A. Bozzon, S. Comai, S.Fraternali, S.Toffetti Carughi. *Capturing RIA Concepts in a Web Modeling Language.* En: International World Wide Web Conference WWW2006 (15, Edinburgh [Scotland], 23-26 de Mayo de 2006).

[Beck03] K. Beck. *Test-driven Deveopment: by example.* Addison Wesley, 1ra Edición, c2003.

[Codepl09] Microsoft, *Codeplex* [en línea] `http://www.codeplex.com` [consultado : 19/11/2009]

[CrHo03] L. Crispin, T. House. *Testing Extreme Programming.* Addison Wesley, 1ra Edición, 2003.

[CubicT10] *CubicTest* [en línea] `http://cubictest.seleniumhq.org/` [consultado : 14/01/2010]

[EGMF09] *Eclipse - GMF - Making Figures Sensitive To Attributes Of Semantic Elements.* [en línea] `http://serdom.eu/ser/2009/01/21/eclipse-gmf-making-figures-sensitive-to-attributes-of-semantic-elements` [consultado : 27/11/2009]

[EMF04] IBM, *The Eclipse Modeling Framework (EMF) Overview*. 2004 [en línea] `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp` [consultado : 27/11/2009]

[FireB09] *Firebug* [en línea] `http://getfirebug.com` [consultado : 07/12/2009]

[FTSF09] Fraternali P., Tisi M., Silva M, Frattini L. *Building Community-Based Web Applications with a Model-Driven Approach and Design Patterns.* [borrador no publicado] [en línea] `http://www.webml.org/webml/upload/ent5/1/CommunityPatterns-final.pdf` [consultado : 10/12/2009]

[Goo09] *Google* [en línea] `http://www.google.com` [consultado : 12/12/2009]

[Gmail09] Google, *Gmail* [en línea] `https://mail.google.com` [consultado : 29/11/2009]

[GMF09] Eclipse, *GMF - Graphical Modeling Framework* [en línea] `http://www.eclipse.org/modeling/gmf` [consultado : 13/12/2009]

[GMFTut09] The Eclipse Foundation, *GMF Tutorial* [en línea] `http://wiki.eclipse.org/GMF_Tutorial` [consultado : 27/11/2009]

[GSV00] N. Güell, D. Schwabe, P. Vilain. *Modeling Interactions and Navigation in Web Applications*. En: Conceptual modeling for E-business and the Web (1, Salt Lake City UT, 9-12 de Octubre de 2000).

[IntP09] *Interaction-Patterns* [en línea] `http://www.interaction-patterns.org` [consultado : 29/11/2009]

[JavaFX09] Sun Microsystems, *JavaFX* [en línea] `http://www.javafx.com` [consultado : 29/11/2009]

[Kbb09] KBB, *Perfect Car Finder* [en línea] `http://www.kbb.com/KBB/PerfectCarFinder/PhotoEdition.aspx` [consultado : 29/11/2009]

[Laszlo09] Laszlo Systems, *OpenLaszlo* [en línea] `http://www.openlaszlo.org` [consultado : 29/11/2009]

[Netfli09] *Netflix* [en línea] `http://www.netflix.com` [consultado : 24/11/2009]

[NVIDIA09] NVIDIA, *Speak Visual* [en línea] `www.speakvisual.com` [consultado : 29/11/2009]

[OGRDC08] L. Olsina, A. Garrido, G. Rossi, D. Distante, G. Canfora. *Web Application evaluation and refactoring: A Quality-Oriented improvement approach*. Journal of Web Engineering, Vol.7 No.4, 1 de Diciembre de 2008.

[PLCS07] J.C. Preciado, M. Linaje, S. Comai, F. Sánchez-Figueroa. *Designing Rich Internet Applications with Web Engineering Methodologies*. En: IEEE International Symposium on Web Site Evolution WSE2007 (9, París, 5-6 de Octubre de 2007).

[PLCS05] J.C. Preciado, M. Linaje, S. Comai, F. Sánchez-Figueroa. *Necessity of methodologies to model Rich Internet Applications*. En: Proceedings of the IEEE International Symposium on Web Site Evolution WSE05 (7 , Budapest, 26 de Septiembre de 2005).

[Quince09] Infragistics, *Quince: UX Pattern Explorer* [en línea] `http://quince.infragistics.com` [consultado : 29/11/2009]

[RIATest09] *RIATest - Test Automation for Adobe Flex* [en línea] `http://www.riatest.com` [consultado : 29/11/2009]

[RRG09a] G. Rossi, E. Robles, J. Grigera. *Bridging Test and Model-Driven Approaches in Web Engineering*. En: International Conference on Web Engineering ICWE2009 (9, San Sebastián [España], 22-26 de Junio de 2009).

[RRG09b] G. Rossi, E. Robles, J. Grigera., Et. al. *Introducing Usability Requirements in a Test/Model-Driven Web Engineering Method*. En: International Workshop on Web-Oriented Software Technologies IWWOST2009 (8, San Sebastián [España], 23 de Junio de 2009).

[RuxTool09] *RUX-Tool* [en línea] `http://www.ruxproject.org` [consultado : 29/11/2009]

[Scott05] B. Scott. *Musings on Mouse Hover*, 2005 [en línea] `http://looksgoodworkswell.blogspot.com/2005/11/musings-on-mouse-hover.html` [consultado : 29/11/2009]

[Scott09] B. Scott. *RIA Patterns - Best Practices for Common Patterns of Rich Interaction* [en línea] `http://www.slideshare.net/interactionpatterns.org/ria-patterns-best-practices-for-common-patterns-of-rich-interaction-presentation` [consultado : 29/11/2009]

[Sel09] *Selenium Web Application Testing System* [en línea] `http://seleniumhq.org` [consultado : 29/11/2009]

[SelFla09] *Flash-selenium - A Selenium extension for enabling Selenium to test Flash components* [en línea] `http://code.google.com/p/flash-selenium` [consultado : 29/11/2009]

[SelFle09] *Selenium-Flex API - Automation for Adobe Flex applications* [en línea] `http://code.google.com/p/sfapi` [consultado : 29/11/2009]

[SelSil09] *Silverlight-selenium - The Selenium RC clients for adding SilverLight Communication Capabilities* [en línea] `http://code.google.com/p/silverlight-selenium` [consultado : 29/11/2009]

[Silv09] Microsoft, *Silverlight* [en línea] `http://www.microsoft.com/SILVERLIGHT` [consultado : 28/11/2009]

[Squish09] *Squish - Cross platform automated GUI and Web testing tool* [en línea] `www.froglogic.com/squish` [consultado : 29/11/2009]

[TEGMF09] The Eclipse Foundation, *The Eclipse Graphical Modeling Framework* [en línea] `http://www.eclipse.org/modeling/gmf` [consultado : 27/11/2009]

[UIPF09] *UI Pattern Factory* [en línea] `http://uipatternfactory.com/` [consultado : 12/12/2009]

[VS02] P. Vilain, D. Schwabe. *Improving the Web Application Design Process with UIDs*. En: Proceedings of the International Workshop on Web Oriented Software Technology (IWWOST'2002) (2, Málaga [España], 10 de Junio de 2002).

[VSdS00] P. Vilain, D. Schwabe, C. de Souza. *A Diagrammatic Tool for Representing User Interaction in UML*. En: Proceedings of the UML2000 Conference (3, York [Inglaterra], 2-6 de Octubre de 2000).

[Watir10] *Watir - Web Application Testing in Ruby* [en línea] `http://watir.com` [consultado : 14/01/2010]

[Walm09] *Walmart* [en línea] `http://www.walmart.com` [consultado : 29/11/2009]

[WebRatio09] *WebRatio Tool Suite* [en línea] `http://www.Webratio.com` [consultado : 29/11/2009]

[Welie09] *Welie - Patterns in interaction design* [en línea] `http://www.welie.com` [consultado : 12/12/2009]

[Wilson09] *Wilson* [en línea] `http://www.wilson.com` [consultado : 29/11/2009]

[WRAjax10] *WebRatio Ajax Extension Demo* [en línea] `http://131.175.57.119/ajaxdemo/sv1.do` [consultado : 31/01/2010]

[WSRIA09] Web Spiders, *Rich Internet Applications - RIA Development* [en línea] `http://www.webspiders.com/rich-internet-applications.aspx` [consultado : 29/11/2009]

[WTDD09] Wikipedia, *Test Driven Development* [en línea] `http://en.wikipedia.org/wiki/Test-driven_development` [consultado : 27/11/2009]

[YFlickr09] Yahoo!, *Flickr* [en línea] `http://www.flickr.com` [consultado : 24/11/2009]

[YMail09] Yahoo!, *Yahoo! Mail* [en línea] `http://mail.yahoo.com` [consultado : 29/11/2009]

[YNews09] Yahoo!, *Yahoo! News* [en línea] `http://news.yahoo.com` [consultado : 24/11/2009]

[YDPL09] Yahoo!, *Yahoo! Design Pattern Library* [en línea] `http://developer.yahoo.com/ypatterns` [consultado : 29/11/2009]

[YUI09] Yahoo!, *Yahoo! User Interface Library (YUI)* [en línea] `http://developer.yahoo.com/yui` [consultado : 29/11/2009]

[XPath09] W3C, *XML Path Language (XPath)* [en línea] `http://www.w3.org/TR/xpath` [consultado : 29/11/2009]

[XPather09] *XPather - XPath Generator and Editor* [en línea] `https://addons.mozilla.org/en-US/firefox/addon/1192` [consultado : 28/11/2009]

Apéndice A

Notación UID

En este capitulo presentamos la notación completa de los diagramas UIDs original [VSdS00] para representar las interacciones entre el usuario y una aplicación.

A.1. Entradas y salidas de datos

Ítem de datos

Un ítem de datos (*data item*) representa una pieza de información simple que aparece durante una interacción. Adicionalmente, se puede especificar el dominio del ítem de datos. El dominio es especificado por el diseñador y en caso de que sea omitido se asume el valor por defecto *Text*.

```
<data item>

<data item>: <domain>
```

Estructuras

Una estructura (*structure*)representa una colección de información relacionada. El contenido de la estructura puede especificarse entre paréntesis [1].

```
<Structure>(<data item1>, <data item 2>.. <data item n>)

<Structure>()
```

Conjuntos

Un conjunto (*set*) representa una colección de estructuras o ítems de datos [2]. La multiplicidad de un conjunto es representada por un rango *min..max* al frente de un ítem de datos o estructura.

```
... <data item>

... <Structure>(<data item 1>, <data item 2>.. <data item n>)

1..5 <data item>
```

[1] Si bien una estructura puede contener otras estructuras o conjuntos, el editor *UID-RIA* limita el contenido de las estructuras a ítems de datos simples.

[2] En el editor *UID-RIA* no existe la entidad conjunto por si misma. En cambio se representan seteando la propiedad multiplicidad (*multiplicity*) de las estructuras e ítems de datos, lo cual permite interpretarlas como conjuntos.

Datos opcionales

Permiten representar ítems de datos, estructuras o textos opcionales. Una pieza de información opcional se representa mediante la adición del símbolo *?*, o mediante la multiplicidad *0..1*. En el caso de una entrada de usuario, se puede representar mediante un rectángulo de línea punteada.

<data item>?

<Structure>(<data item 1>, <data item 2>.. <data item n>)?

Entradas de usuario

Una entrada de usuario (*user input*) representa un ítem de datos o estructura ingresada por el usuario. Toda la información ingresada por el usuario es ubicada dentro de rectángulos.

<data item>

Entradas de usuario enumeradas

Una entrada de usuario enumerada (*enumerated user input*) representa un ítem de datos ingresado por el usuario a partir de una lista de valores dada por el sistema. Los valores provistos por el sistema son enumerados entre corchetes y separados por comas [3].

<data item>[<option1>, <option 2>.. <option N>]

Salidas del sistema

Las salidas del sistema (*system output*) representan ítems de datos o estructuras retornadas por el sistema. Toda la información retornada por el sistema es ubicada directamente en el estado de interacción.

<data item>

<Structure>(<data item 1>, <data item 2>.. <data item n>)

Textos

Un texto (*text*) representa un texto descriptivo mostrado por el sistema que aparece dentro de una interacción.

A.2. Estados de interacción

Un estado de interacción (*interaction state*) representa una interacción entre el usuario y el sistema. La información ingresada por el usuario y retornada por el sistema es mostrada dentro de una elipse que representa al estado de interacción A.1). Un estado de interacción nunca puede estar vacío.

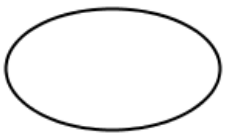

Figura A.1: UIDs - Representación de un estado de interacción

Estado de interacción inicial

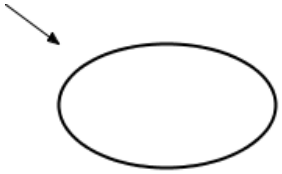

Figura A.2: `UIDs - Representación de un estado de interacción inicial`

Representa el estado de interacción que inicia la cadena de interacciones entre el usuario y el sistema (figura A.2).

Estados de interacción alternativos

Esta representación es utilizada cuando existen dos o mas alternativas de salida de un estado de interacción. Los subsecuentes estados de interacción dependen de la información ingresada o las opciones seleccionadas por el usuario (figura A.3).

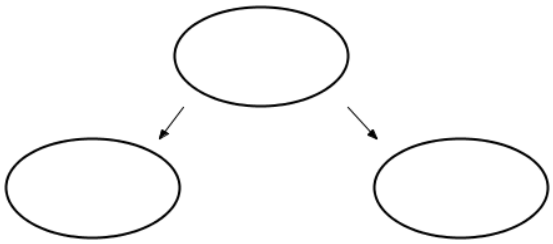

Figura A.3: `UIDs - Representación de estados de interacción alternativos`

Sub-estados de un estado de interacción

Esta representación es utilizada cuando dos partes de un estado de interacción son exclusivas. Las partes exclusivas son ubicadas en diferentes sub-estados (figura A.4).

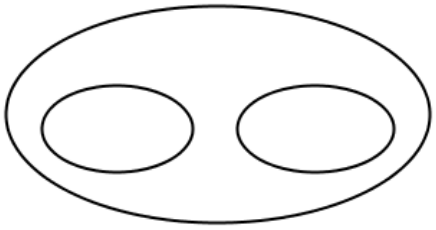

Figura A.4: `UIDs - Representación de sub-estados`

Llamada a otro UID

Esta representación permite describir que el foco de interacción es transferido a otro UID (figura A.5).

Llamada desde otro UID

Esta representación permite describir que el foco de interacción se obtiene desde otro UID (figura A.5).

[3]En el editor *UID-RIA*, no se distinguen las entradas enumeradas de las entradas normales, simplemente al completar la propiedad *options* de una entrada de usuario, se interpreta como una entrada de usuario enumerada.

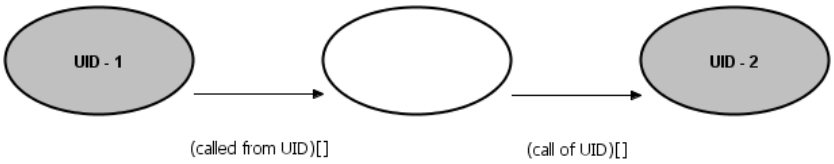

Figura A.5: UIDs - Representación de llamadas desde y hacia otro UID

A.3. Transiciones

Una transición (*transition*) representa que el estado de interacción al que apunta la transición puede convertirse en el nuevo foco de interacción luego que el sistema haya retornado la información necesaria y el usuario haya ingresado los datos requeridos, en el estado de interacción de donde proviene la transición. Existen tres variantes no exclusivas para las transiciones:

Transición con selección de N elementos

Representa que deben seleccionarse N elementos antes de que el estado de interacción al que apunta la transición se convierta en el foco de interacción (figura A.6).

Transición con selección de la opción X

Representa que la opción X debe ser seleccionada antes de que el estado de interacción al que apunta la transición se convierta en el foco de interacción (figura A.6). El nombre de la opción se representa entre paréntesis.

Cuando la selección de una opción no cambia el foco de interacción, la transición puede representarse con una linea y un punto en su final (figura A.7).

Transición con condición Y

Representa que el estado de interacción al que apunta la transición se convertir en el foco de interacción, solo si la condición Y es verdadera (figura A.6). La condición puede expresarse utilizando lenguaje natural.

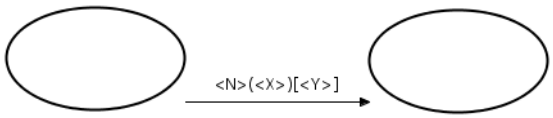

Figura A.6: UIDs - Representación de transiciones

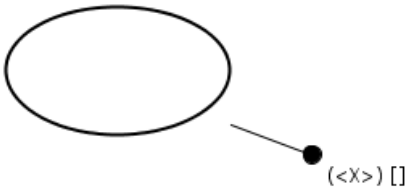

Figura A.7: UIDs - Representación de una transición sin cambio de foco

A.4. Pre y post condiciones

Las pre y post condiciones (*pre and post conditions*) representan condiciones que deben satisfacerse antes y después de que se produzca la interacción representada en el UID, respectivamente.

`Pre-condition: <Y>`

`Post-condition: <Y>`

A.5. Notas textuales

Las notas textuales (textual notes) representan información importante que no puede ser representada gráficamente.

Apéndice B

Modelado RIA en WebML

Como destacamos en el capitulo 6 uno de los puntos que debemos abordar al momento de aplicar nuestra metodología de desarrollo en aplicaciones RIA, es el hecho de ser capaces de modelar aspectos referentes a este tipo de aplicaciones con las metodologías MDWE existentes. Aquí, nos encontramos con dos obstáculos:

1. *La mayoría de las metodologías MDWE cubren pocos o ningún aspecto del modelado de características RIA.*

2. *Las herramientas para el desarrollo RIA son en general de bajo nivel, dependientes de la tecnología y orientadas a código; sin ningún tipo de soporte para el modelado independiente de plataforma y la derivación del código de la aplicación.*

Es por este motivo que surge la necesidad de proponer nuevas metodologías MDWE para el desarrollo RIA o ampliar/modificar las existentes [PLCS05]. En este capitulo describiremos una ampliación del lenguaje de modelado WebML para cubrir ciertos aspectos de las aplicaciones RIA ([BCFT06a], [BCFT06b] y [PLCS07]).

B.1. WebML

El lenguaje de modelado *WebML* (*Web Modeling Languaje*) permite especificar en el plano conceptual la publicación o manipulación de datos de una aplicación Web. El contenido es modelado mediante diagramas de entidad-relación (E-R) o diagramas UML de clase. Sobre el mismo modelo de datos, es posible definir diferentes hipertextos (llamados vistas o *site views*) orientados a diferentes tipos de usuarios o dispositivos de acceso. Un *site view* es un grafo de paginas, posiblemente agrupadas en áreas con un tema en común (por ejemplo, *Amazon* posee el área de libros, el área de música, etc) y jerárquicamente organizadas en sub-páginas.

Las páginas incluyen unidades de contenido, que representan los componentes que publican información: la información que se muestra en una unidad normalmente proviene de una entidad del modelo de datos, y puede ser determinada por medio de un selector, que es un condición lógica que filtra las instancias de las entidades a ser publicadas. Las instancias seleccionadas para mostrarse pueden ser ordenadas de acuerdo a alguna clausula de ordenamiento. Las unidades son conectadas unas con otras a través de enlaces, que trasfieren parámetros y permiten al usuario navegar a través del hipertexto.

El lenguaje *WebML* también permite especificar operaciones que implementan lógica de negocios; en particular, existe un conjunto de operaciones de actualización de datos predefinidas, por

las cuales se puede crear, eliminar o modificar las instancias de una entidad, y crear o eliminar las instancias de una relación.

B.2. Modelado conceptual de aplicaciones RIA

Las tecnologías RIA proveen nuevas capacidades con respecto a las aplicaciones Web tradicionales que deben ser tenidas en cuenta en su diseño. El diseño de aplicaciones RIA mediante el uso de metodologías orientadas a modelos requiere adaptar el flujo de desarrollo Web de las aplicaciones convencionales para considerar las nuevas capacidades disponibles en el lado del cliente, las nuevas características de presentación, y los diferentes flujos de comunicación entre el cliente y el servidor. La figura B.1 muestra las 4 fases que caracterizan al diseño RIA: modelado de datos, lógica de negocios, de presentación, y de comunicación. Estas fases requieren extender los modelos de las aplicaciones Web convencionales.

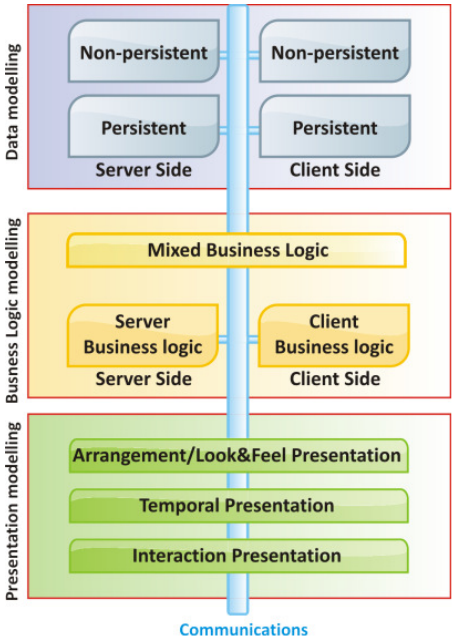

Figura B.1: `Conceptos implicados en el diseño de aplicaciones RIA`

De esta manera, podemos considerar un conjunto común de características de las aplicaciones RIA, con el propósito del modelado conceptual y la generación de código:

- Las aplicaciones RIA deben soportar el procesamiento del lado del cliente y reducir la comunicación con el servidor al mínimo posible.
- Los datos que maneja la aplicación pueden ser almacenados con diferentes niveles de persistencia, tanto en el cliente como el servidor.
- El procesamiento de datos (por ejemplo, la creación, modificación o filtrado de información) puede ocurrir tanto en el cliente como el servidor.
- Es posible la operación en línea y fuera de línea.

Otros requerimientos podrían incluir el soporte para comportamientos de interfaz sofisticados, como *drag and drop*, animaciones, sincronización multimedia y comunicación servidor-cliente.

Las capas usuales de un modelo conceptual Web incluyen el modelado de datos, hipertexto de alto nivel e hipertexto de bajo nivel. La tabla B.1 resume los ítems de modelado afectados, organizados de acuerdo a las capas de un modelo conceptual Web típico. En lo que sigue describiremos las extensiones propuestas a los componentes principales de un modelo conceptual Web, necesarios para soportar características RIA.

Dimensión	Modelado Web tradicional	Modelado RIA
Modelado de datos		
Construcciones	E-R, diagramas UML de clase	E-R, diagramas UML de clase
Nivel de persistencia	servidor (base de datos, sesión)	servidor (base de datos, sesión), cliente
Tipos de relaciones	servidor	servidor, cliente, intra-layer
Modelado de hipertexto de alto nivel		
Construcciones	site views, áreas, páginas	site views, áreas, páginas
Tipos de páginas	páginas en servidor	páginas en servidor y cliente
Modelado de hipertexto de bajo nivel		
Construcciones	unidades de contenido y operaciones en servidor	unidades de contenido y operaciones en servidor y cliente
Selectores	lado servidor	lado servidor y cliente
Ordenamiento	lado servidor	lado servidor y cliente

Cuadro B.1: `Dimensiones de modelado afectadas`

B.2.1. Modelo de datos

Mientras que en las aplicaciones Web tradicionales el contenido reside únicamente en el lado servidor (en la forma de tuplas de una base de datos o como objetos de datos en memoria relacionados a la sesión de usuario); en las aplicaciones RIA el contenido también puede residir en el cliente, como objetos en memoria principal con la misma duración y visibilidad que la aplicación cliente o, en algunas tecnologías, como objetos persistentes del lado cliente. Por este motivo, los datos son caracterizados por dos dimensiones diferentes: la capa de existencia arquitectural, que puede ser el servidor o el cliente, y el nivel de persistencia, que puede ser permanente o temporal.

En *WebML*, donde el modelo de datos es representado con diagramas de entidad-relación, se estereotipan las entidades y relaciones con su nivel de persistencia. La figura B.2 muestra un esquema de datos bien formado. En este caso se estereotipa como *database* a los datos almacenados permanentemente en el servidor (por ejemplo, en una base de datos relacional o XML); como *client* a los datos que son temporalmente almacenados en el lado cliente, durante la duración de la aplicación en ejecución. La figura B.3 muestra un modelo de datos mas completo, en donde se especifica el nivel de persistencia de las entidades.

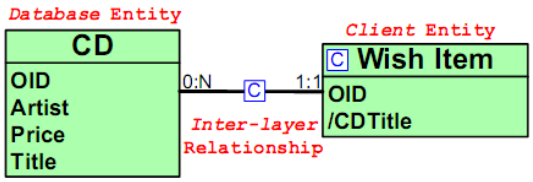

Figura B.2: `Ejemplo de un modelo de datos WebML RIA`

Un esquema de datos extendido con los niveles de persistencia esta bien formado si se cumple la siguiente restricción: las relaciones con persistencia *database* conectan entidades con la misma persistencia (por ejemplo, las relaciones persistentes no pueden conectar entidades temporales).

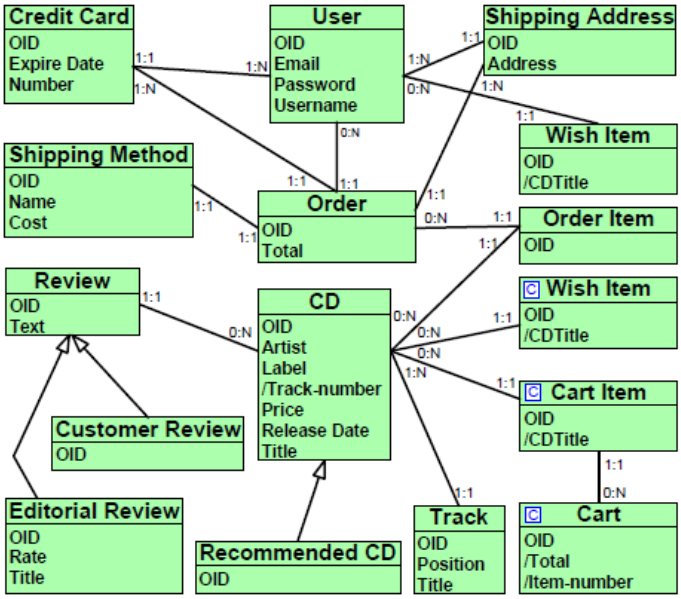

Figura B.3: `Ejemplo de un modelo de datos WebML RIA con especificación de persistencia`

B.2.2. Modelo de hipertexto de alto nivel

El modelo de hipertexto de alto nivel especifica la estructura general del *front-end*: organiza el hipertexto teniendo en cuenta las diferentes clases de usuarios, y estructuras en las páginas, posiblemente agrupadas en áreas que tienen un propósito especifico, y organizadas en una jerarquía compuesta de páginas anidadas. Desde el punto de vista tecnológico las aplicaciones RIA tienen una estructura física diferente a las aplicaciones Web tradicionales: las aplicaciones RIA consisten en general de un único núcleo de aplicación (por ejemplo, una Applet Java o una película Flash), que carga los diferentes datos y componentes en base a las interacciones del usuario. Las aplicaciones convencionales en cambio consisten de múltiples *templates* independientes, procesados por el servidor y simplemente *renderizados* por el cliente. Sin embargo, la metáfora de modelado de hipertexto sigue siendo una buena descripción de la dinámica de las interfaz, también para aplicaciones RIA, especialmente en el caso de aplicaciones híbridas, que se componen de *templates* de paginas tradicionales y componentes RIA. Para contemplar la especificidad de las aplicaciones RIA, donde las páginas, o fragmentos de ellas, pueden ser ejecutados tanto en el lado servidor como en el lado cliente, se extiende la noción de página *WebML* para que las mismas sean estereotipadas como:

1. *Página servidor*: representa una página Web tradicional; donde el contenido y la presentación son calculadas por el servidor, mientras que la *renderización* y la detección de eventos es administrada por el cliente. Los eventos que generan lógica de negocios son procesados en el lado del servidor.

2. *Página cliente*: representa una página que incorpora contenido o lógica procesadas (al menos parcialmente) del lado del cliente. Su contenido puede ser computado por el servidor o el cliente, mientras que la presentación, renderización y manejo de eventos se produce del lado del cliente. Los eventos pueden ser procesados localmente en el cliente o despachados al

servidor.

La figura B.4 muestra un modelo de hipertexto de alto nivel. Gráficamente, las páginas son denotadas por rectángulos, y marcadas con una C o S si son páginas cliente o servidor respectivamente. El significado del ejemplo se aprecia mejor comparándolo con la implementación de la aplicación mostrada en la figura B.5.

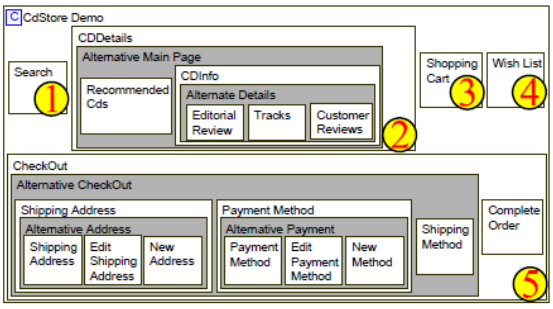

Figura B.4: `Modelo de hipertexto de alto nivel`

Figura B.5: `Rendering del modelo de hipertexto de alto nivel`

B.2.3. Modelo de hipertexto de bajo nivel

El modelo de hipertexto de bajo nivel refina el modelo de alto nivel de la aplicación con detalles sobre el contenido de las páginas, los enlaces para las interacciones de usuario, y las operaciones lanzadas por el usuario. El contenido de las páginas en *WebML* es representado con una notación visual (como un grafo) de unidades de contenido conectadas por enlaces. La figura B.6 muestra un ejemplo de un modelo de hipertexto. Los enlaces expresan tanto el pasaje de parámetros, necesarios para computar los datos de las unidades parametrizadas, como las interacciones de usuario, necesarias para ejecutar la re-computación de la página. En las aplicaciones Web tradicionales, el procesamiento de las unidades de contenido sucede integramente en el servidor: los datos son extraídos desde una base de datos, las condiciones lógicas (llamadas selectores en *WebML*) pueden ser especificadas para filtrar las instancias de las entidades, y las clausulas de ordenamiento especifican como deben ser ordenadas. En las aplicaciones RIA, el computo es distribuido entre el servidor y el cliente, de acuerdo al tipo de las páginas: las unidades contenidas en una página servidor son computadas por el servidor (*server units*), como en las aplicaciones Web convencionales, y las unidades

contenidas en una página cliente son administradas por el cliente (*client units*). El concepto de unidad de contenido de *WebML* es entonces extendido con la posibilidad de especificar la entidad fuente, las condiciones del selector y las clausulas de ordenación ya sea en el servidor o el cliente. Una unidad esta bien formada si cumple con la siguientes restricciones: a) las unidades servidor no pueden ser especificadas sobre una entidad cliente y no pueden comprender selectores y clausulas de ordenación del lado del cliente; b) una unidad cliente que dibuje contenido desde una entidad cliente, no puede contener condiciones de selectores o clausulas de ordenación del lado servidor. Estas restricciones aseguran que todos los cómputos son realizados por el servidor tratando solo con datos y operaciones computables en el lado servidor y así mantener la naturaleza asimétrica de la Web, donde el cliente realiza peticiones al servidor y no viceversa.

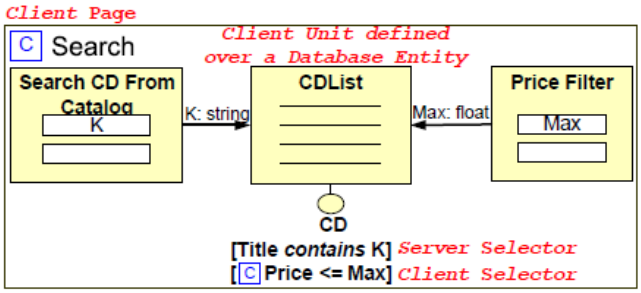

Figura B.6: `Ejemplo de modelo WebML de hipertexto de bajo nivel`

B.2.4. Modelo de operaciones

Las operaciones *WebML* permiten modelar lógica de negocios arbitraria y actualizaciones de contenido predefinidas (creando, eliminando o modificando entidades, conectando o desconectando pares de instancias de entidades pertenecientes a una relación). En el contexto de las aplicaciones RIA, las operaciones pueden ser ejecutadas por el cliente o el servidor, como se describe en la siguiente definición:

1. *Operación de servidor*: es una pieza de lógica de negocios o actualización de datos ejecutada por el servidor.

2. *Operación de cliente*: es una pieza de lógica de negocios ejecutada por el cliente o una actualización sobre una entidad o relación sobre el lado cliente.

3. *Cadena de operaciones*: es una secuencia de operaciones, que puede estar compuesta tanto de operaciones en el lado cliente como el servidor.

La figura B.7 muestra un fragmento de hipertexto para insertar el CD visualizado actualmente en el carrito de compras. La pagina *CDDetails* contiene una unidad de datos que muestra los detalles del álbum actual; el enlace de salida de dicha unidad genera una secuencia de dos operaciones, representadas fuera de la página: *CreateItem* crea una nueva instancia de la entidad cliente *CartItem* y *ConnectToCD* asocia dicha instancia al CD actual, creando una instancia de la relación entre las entidades CD y *CartItem*. Notar que ambas operaciones son marcadas como cliente con una *C*; de esta manera la ejecución de la cadena de operaciones toma lugar completamente en el cliente, sin involucrar al servidor.

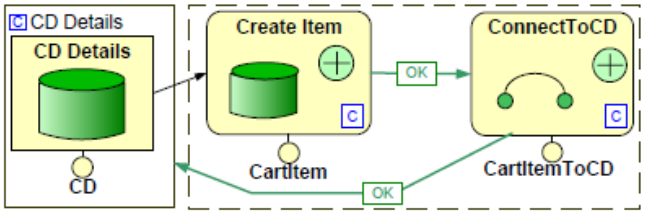

Figura B.7: `Ejemplo de modelado de operaciones en WebML`

B.3. Implementación

Las primitivas discutidas en este capitulo pueden ser implementadas realizando una ampliación del *core* de *WebRatio*. En particular, en [BCFT06a] se describe la forma de adaptar *WebRatio* para soportar estas extensiones y generar el código cliente en la tecnología *OpenLazlo*. La descripción y detalles sobre la implementación de estas extensiones excede a los objetivos de este trabajo.

B.4. Resumen

En este capitulo describimos resumidamente como adaptar el lenguaje de modelado *WebML* para soportar algunas de las características de las aplicaciones RIA. Es importante destacar que estas ideas en realidad tienen un carácter general y no son solo aplicables al uso de *WebML*; la mayoría de las metodologías de diseño Web podrían potencialmente ser extendidas para soportar RIA en una manera similar.