Combinando TDD y MDA para desarrollar aplicaciones Web

Agradecimientos

Son muchas las personas que por un motivo u otro contribuyeron a este trabajo, y a todo el trabajo que lo precedió.

En primer lugar voy a agradecer a mis padres, por ayudarme e incentivar toda mi carrera, de la misma forma que lo hicieron con cualquiera de mis elecciones. Como buenos hijos, mis hermanos los imitaron en esta actitud, así que también voy a agradecer a ellos.

A mis directores Gustavo y Silvia por la incansable motivación y las innumerables oportunidades.

A mis compañeros de estudio junto a quienes logré sortear los primeros años de cursadas, entre otras cosas. Son muchos y todos importantes, pero no puedo dejar de mencionar a mis amigos Cani y Renata.

A Diego Torres (ese no, otro), por muchas cosas que no voy a enumerar por temor a aburrir a algún lector que se haya detenido en esta página.

A los amigos del *boliche* de Multimedia, y la gente del LIFIA en general.

A Andrés "para vos" Fortier, que me ayudó mucho en los primeros pasos dentro de la investigación, y lo sigue haciendo hasta hoy.

A mis compañeros de investigación Esteban y Juan.

También a quienes neciamente estoy olvidando incluir en esta página, e irremediablemente recordaré luego de haber terminado este trabajo.

A todos ellos, muchas gracias.

Índice general

1. Introducción 7
   1.1. Metodologías actuales 7
   1.2. Motivación 9

2. Bases 11
   2.1. Test Driven Development 11
   2.2. Model Driven Web Engineering 15
   2.3. User Interaction Diagrams 15
   2.4. Tests de interacción 16
      2.4.1. Tests Basados en Captura / Reproducción 17
      2.4.2. Tests Programáticos 17
      2.4.3. Tests Híbridos 18
   2.5. Mockups 18
      2.5.1. Mock objects 19
      2.5.2. Paper prototyping 20
      2.5.3. Mockups de presentación 21

3. La metodología en detalle 23
   3.1. Relevamiento de requerimientos 24
      3.1.1. Mockups HTML 25
      3.1.2. Diagramas de interacción de usuario 28
      3.1.3. Tests de interacción 29
   3.2. Desarrollo de prototipo con MDWE 33
      3.2.1. Introducción a WebRatio 33
      3.2.2. Modelado con WebRatio 35
   3.3. Evolución y cambios en los requerimientos 40

4. Experiencia: comparación con metodologías tradicionales 49
   4.1. Trabajo solicitado 50
   4.2. Planeamiento del experimento 50
   4.3. Resultados 52
   4.4. Conclusiones 57

5. Trabajo relacionado 59
   5.1. Trabajos relacionados varios 59
   5.2. Cubic Test 61
   5.3. Behavior Driven Development 62

6. Conclusiones 65
   6.1. Mejoras en la captura de requerimientos 65
   6.2. Mejoras en el manejo de la evolución 66
   6.3. Conciliando metodologías opuestas 67

7. Resultados derivados y trabajo futuro 69
7.1. WebSpec 69
7.2. Usabilidad 73
7.3. Accesibilidad 74
7.4. Manejo de cambios 76
7.5. Trabajo futuro 76

Capítulo 1
Introducción

El desarrollo de aplicaciones web presenta múltiples problemas: la cantidad y frecuencia de los cambios en los requerimientos, junto con la disponibilidad de tiempos de desarrollo cada vez más cortos son dos de los más significativos. Para contrarrestar estos inconvenientes se han propuesto diferentes metodologías de desarrollo que contemplan las modificaciones y evolución de las aplicaciones de una manera controlada.

1.1 Metodologías actuales

Entre las metodologías más comunes que se emplean hoy día en la industria, se destacan las ágiles y las dirigidas por modelos (MDWE – Model Driven Web Engineering). En pocas palabras, las primeras se basan en ciclos muy cortos de desarrollo con frecuentes entregas e intenso contacto con el cliente y sus requerimientos, mientras que las últimas implementan ciclos más largos y de mayor alcance, obteniendo resultados completos en cada iteración.

Las metodologías dirigidas por modelos para aplicaciones web (Koch, Knapp, Zhang, & Baumeister, 2005) (Rossi & Schwabe, 2008) (Ceri, Fraternali, & Bongio, 2000) suelen seguir un proceso de cascada, a través de los siguientes pasos ilustrados en la figura 1.1:

1. Capturar los requerimientos y generar documentación inicial, por ejemplo mediante diagramas de casos de uso (Jacobson, 1992). Cabe destacar que en esta etapa se intenta abordar todos los requerimientos de una sola vez.

2. Construir modelos de dominio, navegación y presentación, en ese orden.
3. Derivar la aplicación utilizando transformaciones de modelos.

No obstante, el modelo de cascada ha sido criticado por su rigidez (Larman, 2003)(Eleftherakis & Cowling, 2003), puesto que no resulta simple introducir cambios en cualquier momento del desarrollo, y al ser la construcción de software un proceso sometido a constantes adaptaciones, adherirse a un modelo semejante puede provocar importantes pérdidas de tiempo antes de que sea posible corregir el rumbo del proceso.

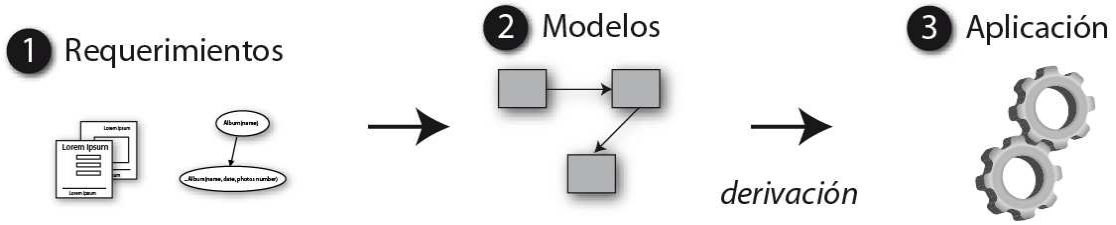


Figura 1.1: Ciclo de desarrollo MDWE

Las metodologías ágiles (Eleftherakis & Cowling, 2003) (McDonald & Welland, 2003) proponen un cambio de enfoque respecto de los modelos de cascada. Siempre se busca seguir ciclos de desarrollo que permitan exponer los resultados a corto plazo a los clientes, e incluso hacerlos participar *durante* el desarrollo mismo por medio de representantes que deben estar disponibles para consultas ante cualquier duda sobre los requerimientos. Se insiste en trabajar en ciclos cortos con tests de aceptación al final de cada uno, para que los clientes puedan verificar los resultados frecuentemente, y así minimizar el riesgo de tomar una ruta equivocada, o alejada de sus necesidades.

La filosofía detrás de las metodologías ágiles se encuentra resumida claramente en el Agile Manifiesto[1], que propone valorar:

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación de contratos.
- Respuesta al cambio sobre seguimiento de un plan.

Test-Driven Development (TDD) se encuentra entre las metodologías ágiles, y como tal respeta estos principios. La idea básica detrás de TDD es escribir tests como una manera de especificar requerimientos antes de programar. De esta manera el desarrollo está guiado exclusivamente por las funcionalidades que se esperan de una versión determinada de la aplicación, y el progreso de las iteraciones es fácilmente mesurable. Los tests además comprueban que todo funcione al cabo de cada iteración, y toman aún más

[1] http://agilemanifesto.org/

valor en iteraciones futuras, donde nueva funcionalidad podría estar interfiriendo con las anteriores. Las iteraciones rápidas y la respuesta al cambio también son características de esta metodología.

En la figura 1.2 se pueden ver los pasos elementales de un ciclo típico de TDD.

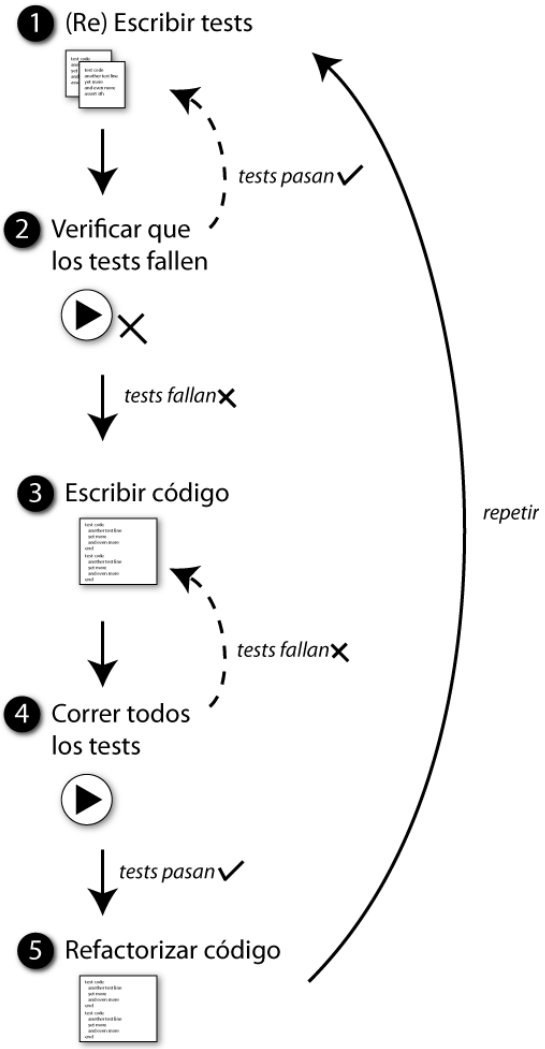


Figura 1.2: Ciclo de desarrollo Test Driven Development

1.2 Motivación

Como se explicó anteriormente, el desarrollo web requiere adaptarse de manera rápida a los frecuentes cambios de requerimientos que típicamente enfrentan este tipo de aplicaciones, pero también se busca que la velocidad de respuesta en el desarrollo no deteriore la calidad del desarrollo existente, comprometiendo esa misma velocidad de respuesta frente a futuros cambios. Las metodologías existentes tienen en cuenta ese requisito y lo contemplan de diferentes maneras.

En el caso de MDWE, al no existir (o estando limitada) la escritura de código fuente dado que las modificaciones se realizan a nivel de modelos, la posibilidad de cometer errores es menor respecto de la escritura de código fuente tradicional, con lo cual la tarea de introducir cambios genera más confianza. No obstante, estas metodologías generalmente no incluyen la escritura de tests como un paso fundamental, cosa que podría traer una mayor tranquilidad a la hora de modificar modelos y generar nuevas aplicaciones, más aún teniendo en cuenta el hecho de que los desarrolladores no tienen por qué conocer el proceso interno de derivación de código, con lo cual se pierde parcialmente la sensación de control sobre el software generado. MDWE tampoco está pensado para realizar iteraciones cortas de desarrollo sobre un sistema, y las herramientas no siempre permiten volver a los modelos una vez derivada la aplicación.

TDD está pensado para lidiar con el cambio. Mediante el uso de tests de unidad, se busca manejar de manera segura y eficiente los cambios de requisitos en el software, pero en lo que respecta a las aplicaciones web, esta metodología sólo puede aplicarse al desarrollo del modelo de dominio (el modelo de negocio subyacente), mientras que gran parte de las aplicaciones web se apoyan fuertemente en la **interacción** y la **navegación**.

La idea de este trabajo consiste entonces en combinar las fortalezas de ambas metodologías para obtener un proceso **robusto** y con buena **capacidad de respuesta** ante los cambios, pero que también considere la navegación y presentación de la aplicación en etapas tempranas del desarrollo. Se pretende con esto crear una metodología de desarrollo de aplicaciones web que cumpla con las siguientes características:

- **Facilidad para establecer requerimientos claros**, que deben ser precisos tanto para los usuarios como para los desarrolladores, tratando a la vez de no generar excesiva documentación que luego agregue mucha carga de mantenimiento, o no sea mantenida en absoluto.
- **Capacidad de respuesta frente a los cambios** con la mayor seguridad posible, para que se puedan poner en producción inmediatamente con un bajo riesgo de errores y sin deshacer funcionalidades previas.
- **Funcionalidad *testeable* en todo momento** no sólo del modelo de dominio, sino de la interacción y navegación, ambos aspectos claves de toda aplicación web.

Teniendo estos objetivos en mente, se describirá una metodología que cubra todas las etapas de desarrollo, pero prestando especial atención a las iniciales (donde se capturan los requerimientos) y a las que tienen que ver con la evolución o los cambios.

Capítulo 2
Bases

La metodología de desarrollo propuesta en este trabajo se basa fundamentalmente en técnicas y herramientas existentes que se emplean actualmente en el desarrollo de aplicaciones web. En esencia, el ciclo de desarrollo es similar al de **TDD**, y como tal también se crearán tests antes de comenzar a desarrollar la funcionalidad, pero en vez de tests de unidad se utilizarán **tests de interacción** que simulan la interacción de un usuario tal como lo haría durante un test de aceptación. Para la etapa de desarrollo propiamente dicho, se optó por emplear herramientas de **desarrollo web dirigido por modelos (MDWE)**. Respecto a la etapa de captura de requerimientos, en la cual pondremos bastante énfasis, se modela la navegación e interacción mediante **Diagramas de Interacción de Usuario (UIDs)** (Vilain, Schwabe, & Souza, n.d.), y otros aspectos de interacción y presentación mediante **mockups** de presentación.

A lo largo de este capítulo se explicarán todas las ideas y herramientas en las que se basó esta metodología.

2.1 Test Driven Development

Dentro de las metodologías ágiles se encuentra **Test Driven Development (TDD)** (Beck, 2002) que combina las técnicas de **Test First Programming** y **Refactoring** (Fowler, Beck, Brant, Opdyke, & Roberts, 1999). En esta metodología, los requerimientos se especifican en forma de tests unidad antes de programar si quiera la primera línea de código. Luego se utilizan para conducir el desarrollo, cuyo único objetivo será el de hacer que dichos tests pasen exitosamente.

Supongamos por ejemplo que tenemos que programar el algoritmo de búsqueda para una agenda de contactos. Al ingresar un término de búsqueda la agenda tiene que hallar coincidencias considerando el nombre del contacto o su teléfono. Podemos expresar este requerimiento con el siguiente test:

```
testSearch(){
      johnMayer := Contact.new("John Mayer", "15 540 0321")
      contactBook.add(johnMayer)

      results := contactBook.search('John')
      self.assertInclusion( results, johnMayer )

      results := contactBook.search('0321')
      self.assertInclusion( results, johnMayer )
}
```

Ahora que tenemos una (vaga) especificación de la funcionalidad de búsqueda, tendríamos que programar el método **search()** para que el test pase. Una implementación posible es la siguiente:

```
search(term){
      contactsFound := Array.new()

      for each contact in self.contacts(){
            if (contact.name().includes(term) or
                contact.phone().includes(term))
            then contactsFound.add(contact)
      }

      return contactsFound
}
```

Con esta implementación, el test pasará sin problemas; sin embargo la búsqueda no funcionará si en vez de buscar a "John" buscamos a "john", o si buscamos "0321 " (con un espacio al final) como teléfono. Visto que el test pasa, el problema no es del desarrollador quien de hecho cumplió su tarea, sino del test que no es lo suficientemente preciso como para contemplar estos casos. Entonces, extendemos el test de la siguiente manera:

```
testSearch(){
      johnMayer := Contact.new("John Mayer", "15 540 0321")
      contactBook.add(johnMayer)

      results := contactBook.search('John')
      self.assertInclusion( results, johnMayer )

      results := contactBook.search('john')
      self.assertInclusion( results, johnMayer )

      results := contactBook.search('0321')
      self.assertInclusion( results, johnMayer )

      results := contactBook.search('0321 ')
      self.assertInclusion( results, johnMayer )
}
```

Con la implementación de actual de **`search()`** el test fallará, así que intentamos hacerlo pasar haciendo algunas modificaciones en el código:

```
search(term){
      contactsFound := Array.new()
      termLowerCase := term.lowerCase()
      termNoSpaces := termLowerCase.trim()

      for each contact in self.contacts(){
         contactName := contact.name()
         contactPhone := contact.phone()
         if (contactName.toLowerCase().includes(termNoSpaces)
              or
            contactPhone.trim().includes(term))
         then contactsFound.add(contact)
      }

      return contactsFound
}
```

Y el test pasa. Pero al adaptar el código perdimos algo de legibilidad, puesto que ahora tenemos la funcionalidad búsqueda junto con la preparación de la cadena de consulta (pasar a minúsculas, eliminar los espacios), todo dentro del mismo bloque de código. Podemos pensar entonces en delegar la búsqueda de coincidencia en el contacto agregando el método **`matches()`**, y quedarnos con la siguiente versión de **`search()`**:

```
search(term){
	contactsFound := Array.new()

	for each contact in self.contacts(){
		if (contact.matches(term))
		then contactsFound.add(contact)
	}

	return contactsFound
}
```

Ahora deberíamos volver a correr el test para cerciorarnos de no haber alterado la funcionalidad de búsqueda mientras mejorábamos el código. Si el test pasa, entonces nos quedamos con la certeza de que la funcionalidad implementada sigue siendo la misma, sólo que ahora contamos ahora con un código más claro.

Podemos concluir entonces que esta manera de desarrollar trae aparejadas dos consecuencias:

- La funcionalidad programada será tan eficiente como la especificación de los tests. Si los tests no son correctos, o lo suficientemente exhaustivos, el software obtenido no funcionará como se espera, aún cuando esos tests pasen correctamente.
- El código fuente puede quedar desprolijo y difícil de mantener. Dado que el único objetivo del desarrollo es el de pasar los tests, no hay restricción alguna acerca de la calidad del código.

El primer punto se resuelve meramente con experiencia: los desarrolladores deben trabajar para escribir tests cada vez más descriptivos y completos. Esta es una práctica que se adquiere con el tiempo, pero que otorga un beneficio a mediano y largo plazo, dado que inculca la práctica de testear todo lo que se programa, aumentando la confianza en el código escrito. Por ejemplo, una manera simple de mejorar gradualmente la calidad de los tests es haciendo que la persona que los escribe no sea la misma que programa para hacerlos pasar. De esta manera, se diluye el riesgo de que un programador escriba tests teniendo en mente la implementación que realizará, lo que podría introducir un sesgo en esos tests. Desde el punto de vista de la Programación Orientada a Objetos, para la cual está dirigida la metodología TDD, esto supone una ventaja: cuando un programador se enfoca en el protocolo de un objeto (o sistema) produce código más cohesivo, lo que lo hace más resistente al mantenimiento dado que el impacto de modificar la implementación de una clase sin alterar su protocolo se reduce notablemente.

El segundo punto (referente a la desprolijidad del código) es el motivo por el cual el refactoring es una herramienta fundamental en TDD. Una vez que el código funciona correctamente y pasa todos los tests, es momento de *limpiar* las desprolijidades para obtener código que sea fácil de extender y modificar. Por ejemplo, podemos aprovechar esta etapa para unificar el código repetido, que es una típica fuente de inconsistencias y complicaciones de mantenimiento. Dado que tenemos tests para todo el código, resultará

muy simple verificar que la funcionalidad no se ha alterado durante la etapa de refactoring.

2.2 Model Driven Web Engineering

Los métodos de ingeniería web dirigida por modelos permiten que los desarrolladores se enfoquen en conceptos de alto nivel en vez de sólo escribir código fuente. Originalmente el concepto surgió para aplicaciones de escritorio, bajo el nombre MDE (Model Driven Engineering – Ingeniería Dirigida por Modelos).

En MDWE, generalmente se especifican diferentes tipos de modelo para la misma aplicación en distintos niveles. Una separación típica es la que considera al modelo de **dominio** aparte de los modelos de **navegación** y **presentación**. Viendo cada uno en detalle:

- El **Modelo de Dominio** está conformado por la lógica de negocio interna. En una aplicación web, este modelo se encarga de resolver las funcionalidades elementales, y los usuarios interactúan con él a través de una interfaz. Este modelo puede ser un simple modelo de datos trabajado con consultas SQL o un sistema orientado a objetos.
- El **Modelo De Navegación** es tal vez el más característico de las aplicaciones web. En él se especifican todos los caminos que puede tomar un usuario mientras interactúa con la aplicación. Típicamente, un modelo de navegación es similar a un grafo, donde los nodos representan páginas y las aristas links. Los UIDs, por ejemplo, son diagramas de navegación (e interacción) que usaremos en la propuesta de metodología de este trabajo.
- Finalmente el **Modelo De Presentación** es el que especifica la interfaz de usuario. Dado que estamos hablando de aplicaciones web, este modelo puede luego ser usado para generar código HTML, películas SWF, etc. El modelo de presentación, sin embargo, debería ser de más alto nivel que los lenguajes concretos. Muchas veces este modelo se omite, y directamente se trabaja con plantillas (templates) HTML, por citar un ejemplo.

Existen herramientas que derivan aplicaciones funcionando a partir de los modelos propios de metodologías como UWE (Koch, Knapp, Zhang, & Baumeister, 2005), OOHDM (Rossi & Schwabe, 2008) o WebML (Ceri, Fraternali, & Bongio, 2000), mediante transformaciones automáticas.

2.3 User Interaction Diagrams

Los UIDs – User Interaction Diagrams (Vilain, Schwabe, & Souza, n.d.) son, como su nombre lo indica, diagramas de interacción de usuario. Fueron ideados para describir el intercambio de información entre un usuario y una aplicación web, y no hacen ninguna suposición sobre aspectos de interfaz de usuario ni el modelo de dominio, por lo que pueden trabajarse en forma completamente independiente.

Un motor importante para la creación de los UIDs fue la necesidad de emplear documentos de requerimientos más concisos que los típicos Use Cases (Jacobson, 1992) o Casos de Uso, los cuales presentan evidentes problemas de mantenimiento, sobre todo en sistemas de gran envergadura donde la cantidad de documentos termina resultando muy difícil de administrar.

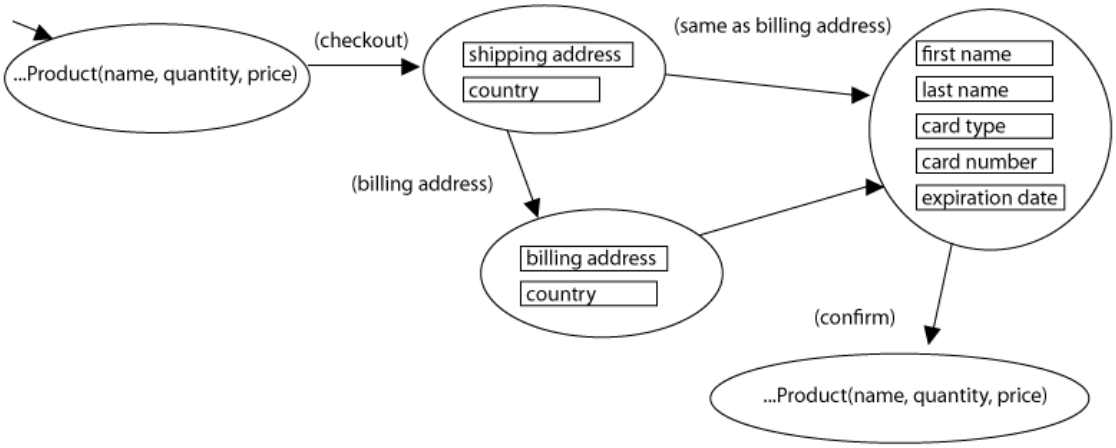


Figura 2.1: Diagrama UID.

La figura 2.1 muestra un diagrama de ejemplo en el contexto de dentro de un sitio de comercio electrónico, donde se especifica la interacción de un comprador al momento de concretar una compra (comúnmente conocido como proceso de *checkout*). Nótese que los nodos contienen la información que completa el usuario, como número de tarjeta o dirección de envío, y las aristas entre los nodos representan las interacciones que realiza el mismo para pasar de un nodo a otro.

Aunque los UIDs sean agnósticos del modelo de dominio, pueden ser muy útiles para comenzar a delinearlo, dado que especifican la información con la cual vamos a tener que lidiar al desarrollar la aplicación.

Cabe hacer la salvedad de que, si bien se mencionó anteriormente que los UIDs se pueden emplear como diagramas de navegación, sus creadores ponen más énfasis en la captura del intercambio de información, e incluso contemplan la posibilidad de emplear UIDs combinados con otros lenguajes de diagramas de navegación en el mismo proceso.

2.4 Tests de interacción

Una vez que un ciclo de TDD (o cualquier metodología ágil) termina, es expuesto a los usuarios para realizar un test de aceptación. En el caso de las aplicaciones web, estos tests van a consistir en navegar por las diferentes partes de la aplicación, llenar formularios, presionar botones, y todas las forma de interacción que uno pueda llevar a cabo con los contenidos que muestra un navegador web. En este punto, una aplicación desarrollada con TDD no va a mostrar fallas en lo que al modelo de dominio respecta, porque todos los tests están pasando exitosamente y además reflejan los requerimientos

de los usuarios. Sin embargo, la satisfacción de los clientes puede estar comprometida por otros motivos, como por ejemplo si la presentación no es la que esperaban o la forma de interactuar no les resulta cómoda. Internamente todo funciona a la perfección, pero los problemas recién mencionados hacen que el modelo de negocio pase a un segundo plano. De hecho, las modificaciones hechas a nivel de navegación o incluso presentación podrían exponer fallas de diseño en dicho modelo de negocio.

Si lográramos capturar los requerimientos de interacción de la misma manera que capturamos aquellos del modelo subyacente a la aplicación, tendría sentido escribirlos en forma de test. Afortunadamente existen varias soluciones para escribir este tipo de tests de interacción. Las herramientas que existen para ello se pueden dividir en dos categorías, según la manera de generar los tests: por un lado los tests basados en captura/reproducción, y por el otro los generados programáticamente.

2.4.1 Tests basados en captura/reproducción (capture/playback)

Esta manera de testear interfaces de usuario consiste básicamente en utilizar la interfaz en cuestión manualmente mientras la herramienta captura la interacción, ya sea detectando qué widgets (elementos de la interfaz) son utilizados, o almacenando directamente los eventos del mouse y teclado, en un nivel más bajo de abstracción. Luego se establecen las aserciones o propiedades que el sistema debe cumplir al cabo de una o varias interacciones.

Este mecanismo presenta varias ventajas: por un lado, **cualquier usuario es capaz de generar un test**, o al menos sus precondiciones. No sería ilógico pensar en hacer que los mismos analistas o incluso los clientes generen las interacciones para poner a prueba el sistema sin el sesgo que consciente o inconscientemente podrían presentar las personas directamente involucradas en el desarrollo (es decir, aquellos que conozcan el funcionamiento interno del mismo).

Otra ventaja importante del modelo de captura/reproducción es que la herramienta de test se mantiene **agnóstica de la tecnología** subyacente al sistema. A diferencia, por ejemplo, de los tests de unidad que suelen utilizar el mismo lenguaje del SUT (System Under Test – Sistema Bajo Test) para poder invocar sus funciones, un test de interacción se centra en simular el comportamiento de un usuario, que por supuesto no tiene por qué conocer los detalles de implementación del sistema que está utilizando.

2.4.2 Tests Programáticos

Esta forma de generar tests de interacción consiste en escribir directamente las acciones que debe realizar el navegador con el sistema en forma de programa, empleando algún lenguaje específico de la herramienta de tests. Esta manera de trabajar resultará mucho más natural para un programador.

La principal ventaja de las herramientas de test programáticas está en que se pueden trasladar todas las prácticas tradicionales de programación a los tests generados, como por ejemplo la posibilidad de modularizar el código, crear tests parametrizables e incluso obtener jerarquías de tests, en el caso que se emplee Programación Orientada a Objetos. Otra ventaja importante está en la simplicidad de integrar los tests con el sistema en sí.

Por último, es importante destacar que con esta técnica **no se requieren mockups** o interfaces existentes para programar un test. Si bien sería de mucha utilidad, no es necesario que existan previo a la escritura, dado que no estaríamos grabando la interacción sino programándola de antemano.

La desventaja evidente de los tests programáticos reside en que sólo los programadores puede escribirlos, y ya no cualquier persona sin este tipo de preparación. Además, puede resultar más lento escribir los tests manualmente que con una simulación, e incluso puede perderse realismo en la interacción, dado que no se está realizando en forma concreta y *en vivo*, sino mediante conjeturas que no se hacen visibles hasta que el test se corre.

2.4.3 Tests Híbridos

Una alternativa interesante es la de combinar los dos tipos de tests para obtener las ventajas de ambos. En un primer momento podríamos grabar los tests con la ayuda de un analista o directamente un cliente, y luego extender dicho test programáticamente, para obtener beneficios como por ejemplo:

- Generar tests similares.
- Crear un modelo de tests, refinando los tests generados con captura y dejándolo preparado para tests futuros.
- Parametrizar el test para aplicar baterías de datos de prueba, y explorar así los límites del rango de posibles datos ingresados.

Por supuesto, necesitamos una herramienta híbrida que permita combinar ambas técnicas. Afortunadamente, estas herramientas existen, y en este trabajo vamos a emplear una de ellas, denominada **Selenium**[2]. Más adelante en la sección 3 se detallarán sus características.

2.5 Mockups

La idea de emplear mockups de presentación está inspirada en la noción de mock objects (Mackinnon, Freeman, & Craig, 2001), empleada en las técnicas tradicionales de tests para programas Orientados a Objetos. En ambos casos tratamos con elementos que por sí solos no tienen comportamiento inteligente, pero sirven para llevar adelante tests que de otra manera serían imposibles de correr. Por otra parte, también el uso de mockups de presentación es similar a varias técnicas de prototipado rápido como paper prototyping (Snyder, 2003).

[2] http://seleniumhq.org/

2.5.1 Mock Objects

Al desarrollar empleando la metodología de TDD tradicional, a menudo nos encontramos con ciertos problemas relacionados con la profundidad del test. Por ejemplo, en el siguiente código de test para la reserva de una butaca de un cine:

```
1 testReservation(){
2     seats := cinema.availableSeats()
3     selected := client.selectBestSeat(seats)
4     cinema.reserve(selected, client)

5     seats := cinema.availableSeats()
6     self.denyInclusion( seats, selected )
7 }
```

El foco del test está en verificar que el cine elimina un asiento del grupo de disponibles luego de que éste sea reservado. En la línea número 2 se obtienen todos los asientos disponibles:

```
...
2     seats := cinema.availableSeats()
...
```

Luego, el cliente elige un asiento entre los asientos libres:

```
...
3     selected := client.selectBestSeat(seats)
...
```

para lo cual se le envía al objeto `client` un mensaje denominado `selectBestSeat(seats: Seat[])`, que selecciona y retorna un asiento del conjunto que se pasa como parámetro, basándose en sus preferencias como cliente del cine. En las líneas siguientes del test se reserva el asiento al cliente, se vuelven a pedir los asientos disponibles y finalmente se verifica que el asiento reservado no está entre ellos:

```
...
4     cinema.reserve(selected, client)

5     seats := cinema.availableSeats()
6     self.denyInclusion( seats, selected )
...
```

Recordemos que el test está centrado en que el cine elimine correctamente un asiento reservado del conjunto de libres, y no lo vuelva a ofrecer. Pongamos ahora atención en el método del cliente que selecciona el mejor asiento entre varias ofertas; esta selección puede resultar muy compleja, pero sin la implementación de este método no podemos llevar a cabo el test de la reserva. Esto es un problema importante si estamos trabajando

con TDD, porque podríamos querer programar el cine *antes* de empezar con los tests del cliente. Para resolver este tipo de problemas existe la noción de **mock object**.

Un mock object puede pensarse como un objeto *de utilería*, que está preparado para responder a un cierto protocolo de mensajes con lo que uno decida, sin la necesidad de programar una clase completa. Por ejemplo, uno podría crear un cliente que siempre responda el primer elemento de la colección de asientos como la mejor selección, o simplemente uno aleatorio. Para el caso del test visto como ejemplo este comportamiento alcanza, dado que no estamos interesados en el mecanismo de selección, y sólo nos interesa contar con que el asiento seleccionado esté dentro de la colección de asientos libres. Esto nos permite enfocarnos en la funcionalidad que queremos verificar sin entrar en detalles ni tener que programar necesariamente los otros objetos involucrados que colaboran con el principal.

La noción de mock object es similar a la de UI mockups respecto a que ambos permiten una solución rápida que permita programar tests antes de tener una aplicación corriendo. En el caso de los UI mockups, generamos interfaces para que los tests de interacción tengan elementos (ya sean de formulario como botones o campos de entrada, o de presentación como títulos, imágenes o párrafos) a los cuales referirse para aseverar predicados acerca de ellos, o permitir la captura de interacción manual.

2.5.2 Paper prototyping

El prototipado de aplicaciones en cualquiera de sus formas sirve para predecir implicancias del desarrollo, pero principalmente para convenir con el cliente acerca de las funcionalidades del sistema, así como de la manera de interactuar con el mismo. En el caso de las aplicaciones web, la interacción y la presentación son partes fundamentales, sobre todo para el cliente, dado que conforman la cara visible de la aplicación. A partir de estos factores podemos llegar a determinar qué funcionalidades van a ser más o menos utilizadas, e incluso descubrir funcionalidades nuevas. Así es que una falta de previsión a nivel de presentación e interacción puede conllevar un desarrollo costoso de funcionalidades que van a ser alteradas radicalmente, que no van a ser utilizadas en absoluto, o que directamente están faltando.

Las ventajas del prototipado son evidentes, pero el prototipado de aplicaciones puede llegar a ser muy costoso, o insumir mucho tiempo. Por supuesto, las características del prototipo que generemos van a estar directamente ligadas con las ventajas o la previsibilidad que aportará al desarrollo. Existen maneras de aprovechar el trabajo hecho para un prototipo, por ejemplo haciéndolo evolucionar hasta que se convierta en la aplicación final, de manera que ningún esfuerzo es descartado completamente.

Una alternativa atractiva para hacer que el prototipado no sea costoso y aún así obtener ventajas significativas de él es la de emplear la manera más barata posible: prototipado en papel. En su libro (Snyder, 2003), Carolyn Snyder explica las bondades de este modelo de prototipado, que consiste simplemente en dibujar borradores de la futura interfaz en papel y presentarlo a los potenciales usuarios, simulando la interacción, incluso empleando formularios y navegación entre páginas. Lo bueno del prototipado en papel es que se desarrolla muy rápidamente, y que las ventajas que otorga en comparación con el esfuerzo que requiere son importantes.

En la misma línea del prototipado rápido, existen herramientas comerciales para realizar prototipos del estilo informal de paper prototyping como Balsamiq[3] o Axure[4]. Estas herramientas permiten generar prototipos de una manera rápida y con las mismas ventajas del prototipado en papel, e incluso permiten en algunos casos comenzar a delinear la presentación final a partir de los prototipos generados.

2.5.3 Mockups de presentación

Los mock objects son objetos sin inteligencia, es decir, sólo hacen lo necesario para que un test pueda ser llevado a cabo. La misma idea rige para los mockups de presentación o GUI Mockups. Si, siguiendo las prácticas de TDD quisiéramos generar un test de captura/reproducción para una aplicación web que estamos por desarrollar, nos encontraríamos con una traba bastante obvia: no podemos grabar la interacción con una interfaz que no existe. Para remediar este problema, necesitamos crear una interfaz que nos permita al menos interactuar con paneles, botones o cualquier elemento visual para poder simular una interacción. Cabe destacar que, si bien en la mayoría de los casos podemos interactuar con una interfaz web sin necesidad de un servidor, existen excepciones a estos casos. Una excepción común se da cuando la interfaz muta en el medio de una interacción sin pasar a otra página, como sucede en el caso de las aplicaciones RIA (Rich Internet Application) enriquecidas con Ajax. Una alternativa para crear tests contra este tipo de interfaces está en validar programáticamente las nuevas versiones de la interfaz a medida que se van generando los cambios.

Los mockups de presentación surgieron entonces como una necesidad para poder crear tests de captura/reproducción antes de que una determinada interfaz (y la funcionalidad a la que permite acceder) exista aún. Creando simples maquetas HTML es posible generar las **precondiciones** de casi cualquier test de interacción, dado que en la mayoría de los casos no se necesita comportamiento del lado del servidor para interactuar. Posteriormente hay que completar estas precondiciones, que corresponderían con la interacción en sí con las **aserciones** acerca de los resultados. Por ejemplo, un test que verifique el funcionamiento de un formulario de registro en un sitio de compras online podría ser esquemáticamente de la siguiente manera:

1. Ingresar el texto "James Hendrix" en el campo con id "name".
2. Ingresar la fecha "27/11/42" en el campo con id "birthday".
3. Ingresar "blackbeauty" en el campo "password".
4. Ingresar "blackbeauty" en el campo "repeat_password".
5. Hacer click en el botón con id "submit".
6. Esperar hasta que la página cargue.
7. Verificar que se muestra el texto "Registration successful" en la página resultante.

[3] http://www.balsamiq.com/
[4] http://www.axure.com/

Aquí, los cuatro primeros pasos se corresponden con una interacción del usuario, es decir las precondiciones, mientras que el último paso consiste en una verificación de los resultados de la interacción, es decir una aserción.

Otro beneficio importante de emplear los mockups de presentación/interacción está en que obtenemos gratuitamente una interfaz que puede ser considerado como la interfaz final de la aplicación, que por supuesto va a requerir de un trabajo estético, pero que si se mantiene esencialmente igual en términos de interacción no va a traer sorpresas al momento de presentarlo a los clientes luego del desarrollo de la funcionalidad, dado que resultará familiar gracias a la presentación previa en la etapa de la generación de los tests de interacción.

Capítulo 3
La metodología en detalle

A lo largo de esta sección se explicará la metodología de desarrollo paso a paso. La figura 3.1 ilustra todas las etapas de la misma. Como se puede apreciar en la figura, se trata de un ciclo iterativo, en el cual se obtienen prototipos que incrementalmente satisfacen los conjuntos de requerimientos tratados durante cada iteración.

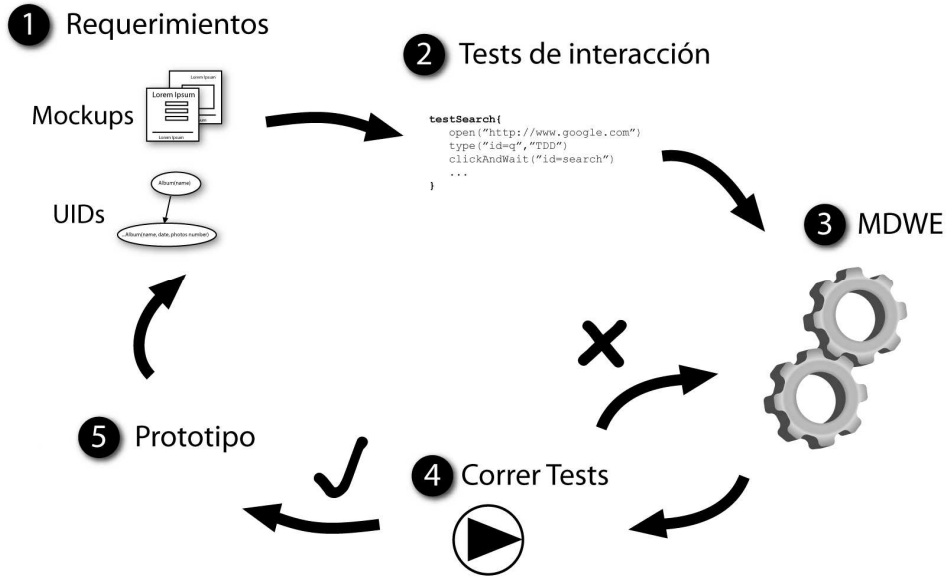


Figura 3.1: Esquema general de la metodología.

En la primera etapa (item 1 en la figura) se toman los requerimientos mediante UIDs y Mockups de presentación. También se pueden usar artefactos tradicionales, como Use Cases o User Stories (Jeffries, Anderson, & Hendrickson, 2000), pero su uso es opcional y no será cubierto en este trabajo. La generación de tests de interacción (2), si bien está especificada aparte, también puede considerarse como parte de la captura de requerimientos dado que modelan de manera precisa (de hecho, pueden ejecutarse) los requerimientos de interacción. En un principio, los tests se corren sobre los mockups de presentación y claramente fallarán, lo cual es esperable y da pie a la etapa de desarrollo (3). Durante esta etapa empleamos MDWE para poder crear la funcionalidad necesaria y hacer que los tests de interacción pasen, para lo cual es preciso adaptar la presentación del prototipo resultante para que coincida con los mockups. Luego del desarrollo, los tests de interacción se vuelven a correr (4) para comprobar que la funcionalidad está correctamente implementada. Si los tests fallan, se continúa con el desarrollo hasta que pasen, y de esta manera obtengamos un prototipo completo (5) con la funcionalidad cubierta en los requerimientos que se abordaron en este ciclo. Recordemos que la idea es siempre tomar unos pocos requerimientos y llevar a cabo un ciclo completo, para luego ir sumando más conjuntos de requerimientos hasta completarlos a todos.

Para ilustrar mejor el proceso, además de explicar en qué consiste cada etapa, se presentará el desarrollo de una aplicación social simplificada similar a Facebook[5] a la que llamaremos PhotoShare. En esta aplicación existirán usuarios que puedan cargar fotos en sus cuentas, organizadas en álbumes. Los usuarios podrán también ver álbumes de otros usuarios, agregándolos antes como amigos. Concretamente, las funcionalidades que se mostrarán para ejemplificar el desarrollo son:

- Registro de un usuario nuevo.
- Login y página principal.
- Administración de álbumes y fotos.
- Agregado de amigos.

A través de las siguientes subsecciones veremos cada etapa del ciclo de desarrollo junto con las herramientas empleadas, a la vez que se ejemplificará cada una de ellas con requerimientos de PhotoShare.

3.1 Relevamiento de requerimientos

Al momento de relevar requerimientos, la metodología busca principalmente dos objetivos: por un lado intenta obtener una comunicación clara con el cliente, tratando de conseguir respuestas inmediatas y exponiendo prototipos para despejar dudas sobre la funcionalidad, presentación e interacción. Por otro lado se busca obtener un conjunto de requerimientos precisos para el equipo de desarrollo a la misma vez. Tratar de satisfacer estos dos requisitos conjuntamente puede resultar complicado, pero es vital para acortar la distancia entre lo que el usuario pide para la aplicación y lo que los desarrolladores entienden que deben programar. A su vez, acortar esta distancia es una buena manera de

[5] http://www.facebook.com/

obtener respuesta rápida del cliente en todas las etapas del desarrollo, incluso las más tempranas.

Para llevar a cabo lo que se pretende para la etapa de requerimientos, existen tres herramientas que ayudan en gran medida para cubrir, por un lado, la necesidad de una buena comunicación, y por el otro la necesidad de tener especificaciones claras que se correspondan con las necesidades del usuario. Las herramientas puntualmente son:

1. Mockups HTML
2. Diagramas de interacción de usuario (User Interaction Diagrams – UIDs)
3. Tests de interacción

Las tres combinadas pueden ayudar en buena medida para la etapa de relevamiento, no sólo guiando el desarrollo sino además dejando una documentación clara, compacta, y fácil de mantener.

3.1.1 Mockups HTML

Los mockups HTML son una posible implementación de los UI Mockups explicados en la sección 2. Técnicamente, un mockup HTML no es más que una página HTML estática sin comportamiento (pero que muestra información de ejemplo que será generada dinámicamente en la aplicación final). Lo característico de un mockup no es entonces su composición sino el uso que se le da.

Cuando un cliente nos explica qué quiere de su aplicación web, hay muchas formas de tratar de acordar las ideas, ya sea en forma de texto, dibujando bocetos en papel o realizando diferentes diagramas. Una alternativa simple, concreta y cercana a una aplicación web es la de directamente prototipar páginas HTML estáticas que contengan lo necesario para establecer una interacción determinada.

Consideremos que iniciamos el desarrollo de la aplicación de ejemplo a partir de la funcionalidad de la página de inicio. El usuario explica entonces que la página principal debe tener las últimas fotos de los usuarios amigos, junto con un link hacia los álbumes propios del usuario y otro para crear un álbum nuevo. Sin mayores especificaciones, podríamos crear un mockup básico con estos requisitos como el que se muestra en la figura 3.2.

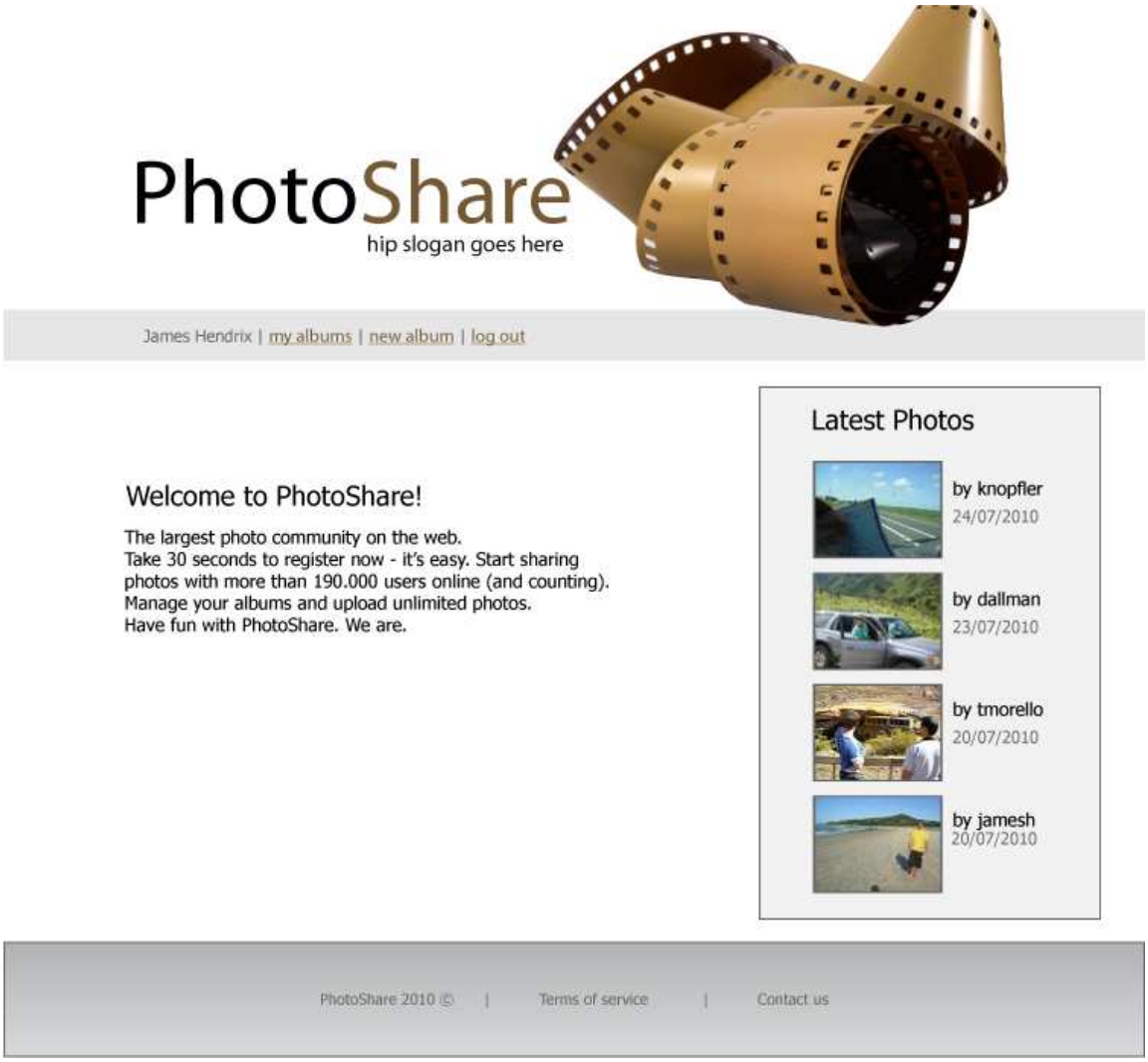


Figura 3.2: Mockup de presentación para PhotoShare.

Si bien el mockup no permite ninguna interacción y los links no conducen a ningún sitio, sirve como base concreta y tangible para especificar cómo debe funcionar la aplicación en este punto.

Siguiendo con los requisitos, podríamos avanzar hacia la administración de álbumes. El usuario nos explica que para un usuario registrado tiene que existir la posibilidad de listar sus álbumes de fotos, crear álbumes nuevos y cargar nuevas fotos en ellos. En este caso tenemos que modelar una interacción algo más compleja que un simple recorrido de navegación. Por ejemplo, un mockup para la vista de un álbum con la capacidad de cargar fotos nuevas podría verse como en la figura 3.3.

Figura 3.3: Mockup de un álbum de PhotoShare.

Aquí estamos mostrando un listado de fotos junto con información general del álbum. Además, hay lugar para cargar nuevas fotos. Este formulario representado en el mockup da una clara idea de lo que tiene que suceder cuando el usuario complete la información requerida (que en este caso consiste en cargar un archivo de imagen y completar un epígrafe) y finalmente presione el botón titulado "Agregar". Sin embargo, ese es el límite de lo que se puede representar mediante un mockup de presentación, dado que la funcionalidad pretendida se especificará en forma de test de interacción como se verá en la sección siguiente. No obstante, la especificación de tests es posible gracias a que el mockup brinda el marco para realizar las interacciones que oficiarán de precondiciones, con lo cual la generación de mockups es vital y se complementa con la generación de tests de interacción.

Más adelante se detallará cómo los mockups no sólo sirven para convenir *look&feel* con los clientes, sino que tienen el valor adicional de permitir especificar tests de interacción del estilo capturar y reproducir.

3.1.2 Diagramas de interacción de usuario

Para modelar y documentar la interacción que tendrá la aplicación, usaremos UIDs. Esta especificación se hace en un nivel alto de abstracción, dado que en este punto no nos hemos interiorizado aún con la tecnología que se empleará para desarrollar la aplicación.

Volviendo a nuestra aplicación de ejemplo, podemos ver en la figura 3.4 un diagrama UID para el registro de un usuario nuevo del sistema:

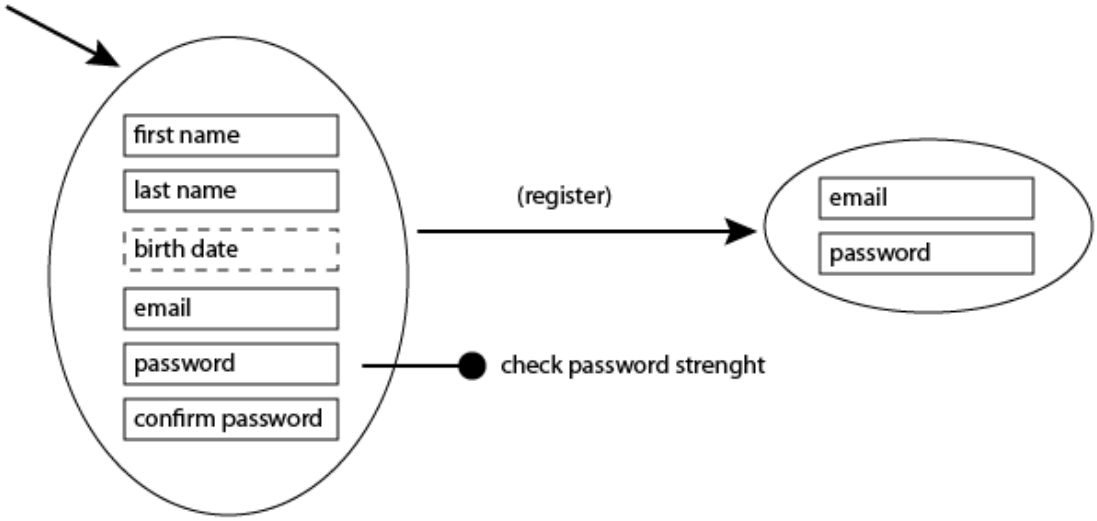


Figura 3.4: Registro de un Usuario en PhotoShare.

Dado que los UIDs representan el intercambio de información, este es el primer punto del desarrollo donde podemos vislumbrar las entidades que formarán parte de nuestro diseño y la información que contienen. Además, servirá como una buena base para especificar los tests de interacción, dado que pueden obtenerse también indicios acerca de la navegación durante el intercambio de información descripto.

Considerando la figura 3.4 en el marco de la aplicación PhotoShare, ya podemos extraer la entidad que representa a un **usuario** como candidata a ser creada en la etapa de desarrollo con modelos, y viendo la información que se ingresa para el registro, podemos también asumir que un usuario tendrá un nombre y un apellido, una fecha de nacimiento (aunque posiblemente no, dado que en un diagrama UID el borde de línea discontinua implica que la información es opcional) un email y una contraseña. Si bien en el intercambio de información vemos que hay un campo más para que el usuario repita la contraseña, esto no necesariamente se traducirá en un atributo dentro del modelo. Aunque esto último pueda parecer un detalle, es un claro ejemplo de la potencial diferencia que puede haber entre la representación de intercambio de información entre el sistema y el usuario (que es el propósito de los diagramas UID) y la información que concretamente se representará en la aplicación como parte del modelo de negocio. En este punto, podría marcarse una analogía entre las transiciones de los UIDs y el modelo de navegación, pero en este caso es más sutil.

Siguiendo con la extracción de información a partir del UID, de la misma manera que detectamos a la entidad usuario junto con sus atributos, a partir de otros UIDs relacionados con la creación y administración de álbumes de fotos, podremos extraer las entidades **álbum** y **foto** junto a sus respectivos atributos, facilitando la tarea de diseño.

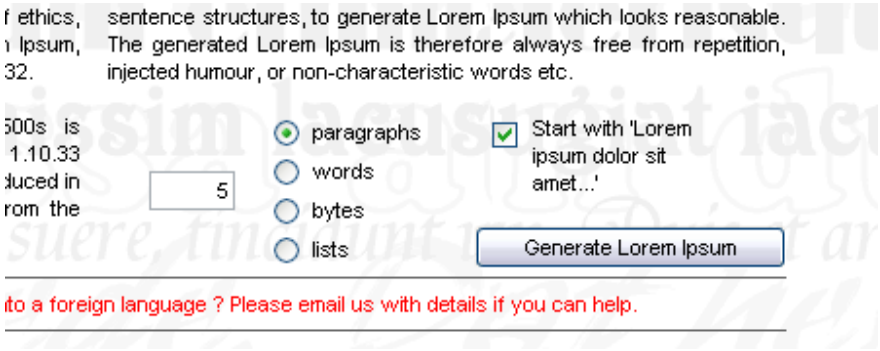
También en el marco de la aplicación, pero ya en vistas de la generación de tests de interacción, podemos validar que la navegación que se intenta realizar en ellos tenga una coincidencia con lo que se especifica en los UIDs diagramados, o que al menos se pueda verificar que todos los caminos posibles contemplados durante el intercambio de información estén transitados por al menos un test de interacción, dado que, como se explicó anteriormente, no necesariamente hallaremos una coincidencia uno a uno entre todos los caminos del intercambio de información y los caminos de navegación definitivos de la aplicación.

3.1.3 Tests de interacción

Para que los requerimientos puedan especificarse concretamente, los tests de interacción cumplen un papel fundamental. Mediante la simulación de la interacción del usuario automatizable, es posible reproducir el comportamiento esperado por el cliente en cualquier momento. Esto es de vital importancia a la hora de desarrollar, dado que el trabajo se concentrará sólo en las funcionalidades solicitadas, de la manera en que fueron solicitadas. Si se siguen los tests como guías de desarrollo, se minimizan

- las especulaciones sobre la interacción.
- la diferencia que ve el cliente entre lo que pidió y el producto entregado.

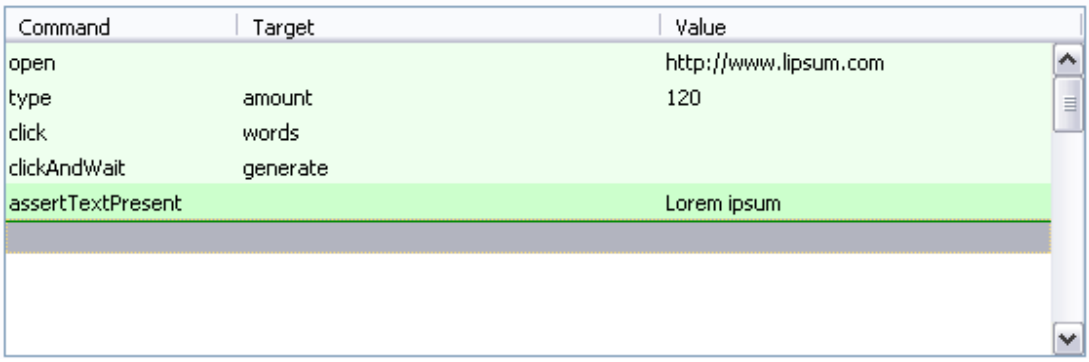
Para crear tests de interacción existen varias herramientas. En nuestro caso elegimos Selenium por su comodidad a la hora de especificar tests y exportar código, y porque funciona además en todos los navegadores principales del mercado (Internet Explorer, Firefox, Safari y Chrome de Google). Selenium funciona tanto como herramienta de *record/playback* como para desarrollar directamente escribiendo código en diferentes lenguajes. El primer caso nos permite abrir una aplicación con un navegador, presionar "record" y comenzar a interactuar. Toda interacción realizada mientras el botón de "record" esté encendido (incluso las esperas) será registrado por Selenium y volcado en forma de código. Si detenemos el grabador y guardamos la interacción, podemos volver a cargarla luego y veremos que todo lo que hicimos anteriormente se repite automáticamente. Veamos un ejemplo de código para un test en el cual se completa un formulario para generar texto aleatorio del siguiente formulario del sitio www.lipsum.com:



y el correspondiente test

Command	Target	Value
open		http://www.lipsum.com
type	amount	120
click	words	
clickAndWait	generate	

Una vez que la interacción está capturada, podemos agregar aserciones acerca del comportamiento de la aplicación, como por ejemplo para verificar que entre el texto que generamos está presente la expresión "Lorem ipsum" tal como lo requerimos en el formulario.



Command	Target	Value
open		http://www.lipsum.com
type	amount	120
click	words	
clickAndWait	generate	
assertTextPresent		Lorem ipsum

Una vez que el test está completo podemos exportar a código Java (entre muchos otros lenguajes), lo que nos permite desarrollar otros tests similares sin la necesidad de capturar una interacción en tiempo real, lo que ahorra mucho tiempo y hace posible que generemos tests exhaustivos para un conjunto de interacciones similares.

```
public void testGenerate() throws Exception {
      selenium.open(""http://www.lipsum.com/");
      selenium.type("amount", "120");
      selenium.click("words");
      selenium.click("generate");
      selenium.waitForPageToLoad("30000");
      assertTrue(selenium.isTextPresent("Lorem ipsum"));
      }
```

En el ejemplo concreto de nuestra aplicación PhotoShare, podríamos tomar alguno de los mockups presentados en la sección 3.1.1 para simular la interacciones que se pueden iniciar a partir de las páginas representadas y validar que los resultados de dichas interacciones condigan con lo representado en los UIDs.

Vayamos por ejemplo a la funcionalidad de crear álbumes de fotos. En la figura 3.5 vemos el mockup HTML generado para esta funcionalidad.

Figura 3.5: Mockup para la creación de álbumes en PhotoShare.

Puntualmente en el mockup podemos ver que para la creación de un álbum se pautó que el usuario debe ingresar un nombre y una fecha. Luego de haber ingresado dicha información, al presionar el botón "Create" deberíamos pasar a la página que muestra los detalles del álbum, que a su vez es la misma que permite cargar fotos. En la creación de un álbum es importante asegurar:

- que la navegación hacia la página de información del álbum sucede.
- que en la página de información figuran el nombre y la fecha especificados en el formulario de creación.
- que el listado de fotos está vacío.

Teniendo en cuenta la interacción que acabamos de describir y las condiciones que deben mantenerse, podemos generar un test para la creación de un álbum, como se muestra en el código siguiente:

```
1.  public void testCreateAlbum() throws Exception {
2.    selenium.open("http://localhost:8080/PhotoShare/page2.do");
3.    selenium.type("name", "Vacation");
4.    selenium.type("date", "11-12-2009");
5.    selenium.click("create");
6.    selenium.waitForPageToLoad("30000");
7.    assertTrue(selenium.isTextPresent("Vacation"));
8.    assertTrue(selenium.isTextPresent("11/12/2009"));
9.    assertTextNotPresent("<ul id=”photos”>");
10.
11. }
```

El test primero lleva a cabo la interacción necesaria para crear el álbum de la misma forma que lo haría un usuario. De hecho, las primeras líneas (desde la 2 hasta la 6) fueron grabadas por Selenium durante una interacción real. El resto de las líneas (desde la 7 hasta la 10) son las que contienen el código que verifica que las postcondiciones especificadas se cumplan. Por ejemplo, la línea 7 valida que luego de crear el álbum, en la página que se carga se esté mostrando el título del álbum ingresado en la línea 3:

```
6. ...
7. assertTrue(selenium.isTextPresent("Vacation"));
8. ...
```

Para validar esto se usa la función Selenium `isTextPresent(...)`, que verifica que el texto que se envía como parámetro esté presente en la página.

3.2 Desarrollo de prototipo con MDWE

El desarrollo con modelos tiene varias ventajas. En primer lugar, evita los errores típicos de la codificación manual y simplifica la extensión. Por estos motivos la metodología se pondrá a prueba con MDWE como alternativa de desarrollo, si bien podría funcionar con otras técnicas aprovechando de manera similar las ventajas de aplicar un ciclo dirigido por tests de interacción.

Como se mencionó en secciones anteriores, la herramienta empleada para probar la metodología será WebRatio[6], una herramienta CASE basada en el lenguaje WebML (Ceri, Fraternali, & Bongio, 2000) para desarrollar aplicaciones web a través de modelos.

3.2.1 Introducción a WebRatio

De los tres posibles modelos explicados en la sección 2.2 contemplados por las metodologías dirigidas por modelos, específicamente modelo de datos, modelo de navegación y modelo de presentación, WebRatio permite representar todos, aunque con diferentes niveles de abstracción.

El **modelo de datos** de WebRatio consiste directamente en Diagramas de Entidad-Relación clásicos, que se transforman en un modelo relacional con una base de datos de soporte. La figura 3.6 muestra un modelo de datos para una aplicación de venta de libros online. En él se pueden ver las diferentes entidades que participan, como libro, orden de compra o usuario (**Book**, **ShoppingOrder** y **User** respectivamente), así como las relaciones entre ellas. También se puede ver la entidad que representa al carrito de compras (**VirtualCart**) en un sombreado diferente, que indica que se trata de una entidad volátil (cuya función se explicará más adelante).

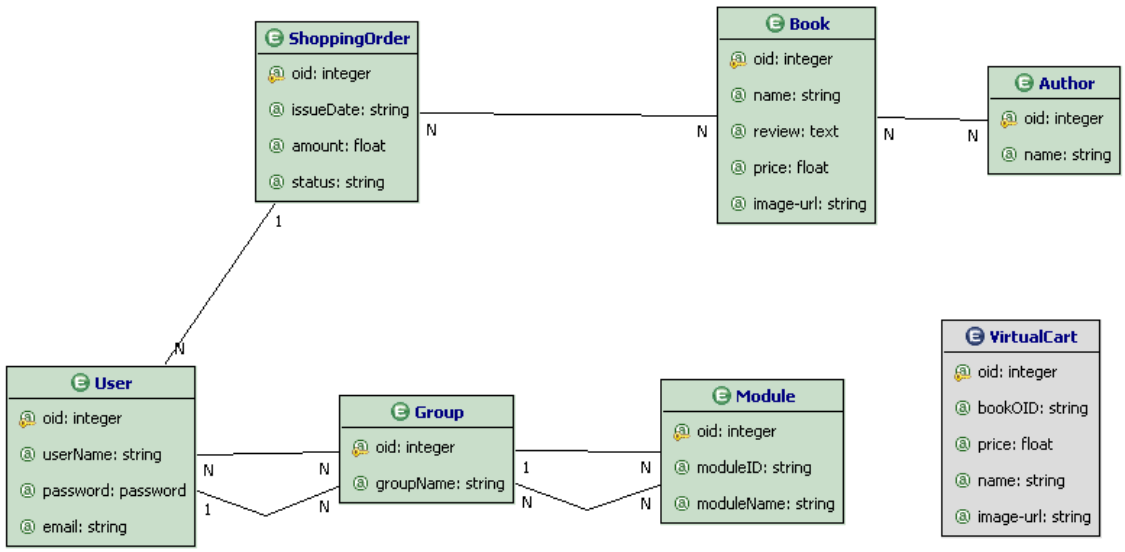


Figura 3.6: Modelo de datos de WebRatio

[6] http://www.webratio.com

El **modelo de navegación** es la característica más fuerte de la herramienta: basado en el lenguaje WebML, permite describir interacciones entre unidades, módulos, páginas y todas las nociones del dominio web soportadas por dicho lenguaje. Además, al ser el modelo de datos un modelo Entidad Relación, toda la lógica que no es sólo navegación también se modela en este punto. Este modelo se traduce automáticamente clases Java y archivos de configuración necesarios para correr la funcionalidad de la aplicación: lectura y escritura de datos, transacciones, procedimientos, alertas, cálculo, etc. En la figura 3.7 se muestra un modelo de navegación para la misma aplicación a la que pertece el modelo de datos de la figura 3.6.

Además de la representación de elementos de presentación o ligados a las fuentes de datos, WebRatio permite establecer conexiones con sistemas o fuentes de datos externas, como por ejemplo Web Services.

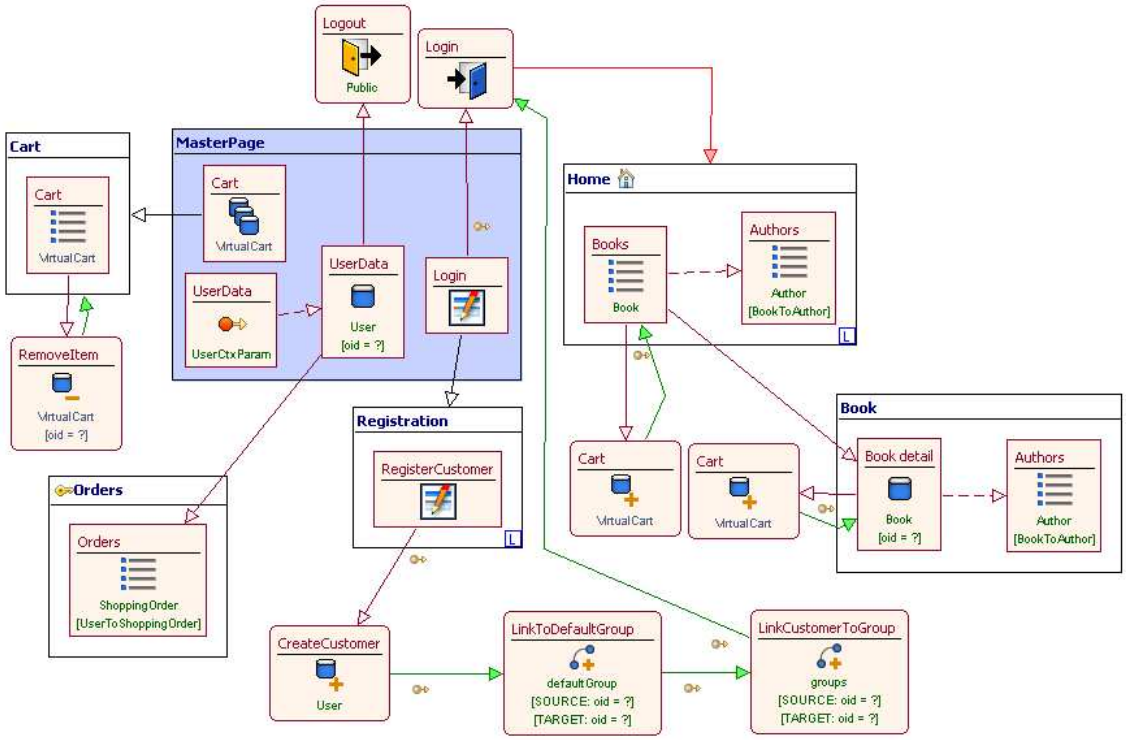
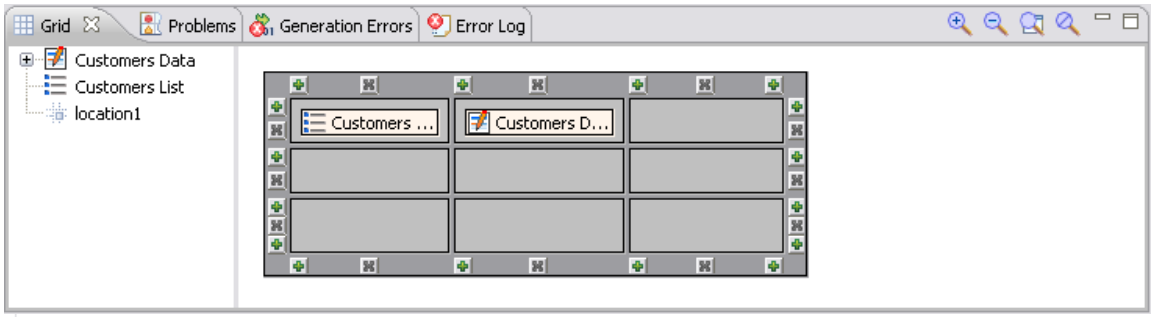


Figura 3.7: Modelo WebML en WebRatio

Finalmente, el **modelo de presentación** se especifica directamente mediante templates con código HTML y fragmentos dinámicos en lenguaje propietario de WebRatio junto con scriplets Groovy[7]. Este modelo es el más rudimentario de los tres; si bien permite hacer algunos ajustes de layout gráficamente en plantillas con forma de grilla, el grueso del trabajo de diseño de presentación para una aplicación real consiste en la escritura manual de código, que además involucra varios lenguajes. La transformación de este modelo genera páginas dinámicas jsp que completan la aplicación creada por el modelo de navegación.

[7] http://groovy.codehaus.org/




```
<webratio:Page page="<wr:PageId/>"/>
<html>
 <head>
    <title> TDDStore - buy books </title>
    <link type="text/css" rel="stylesheet" href="css/basic.css" media="screen"></link>
    <link type="text/css" rel="stylesheet" href="css/product-list.css" media="screen"></link>
    <link type="text/css" rel="stylesheet" href="css/product-detail.css" media="screen"></link>
    <link type="text/css" rel="stylesheet" href="css/register.css" media="screen"></link>
    <link type="text/css" rel="stylesheet" href="css/orders.css" media="screen"></link>
    <link type="text/css" rel="stylesheet" href="css/cart.css" media="screen"></link>
    <!--[if IE]>
        <link type="text/css" rel="stylesheet" href="css/IEBugs.css" media="screen"></link>
    <![endif]-->
  <wr:LinkedResources/>
</head>
<body>
<wr:PageForm>
    <div id="main">

        <div id="header">
            <h1> TDDStore </h1>
            <div id="login-data">
                <wr:CustomLocation name="login-data"/>
            </div><!-- login-data -->
            <script type="text/javascript" src="js/style-functions.js"></script>
```

Figura 3.8: Modelo de presentación y templates de WebRatio

En la figura 3.8 podemos ver las dos caras del modelo de presentación de WebRatio. En la parte superior se ve la grilla que permite modificar la disposición de las componentes gráficas simplemente arrastrándolas con el mouse. Esto generará una tabla HTML en la presentación final, aunque también se permite crear casilleros personalizados fuera de esta grilla. En la parte inferior de la figura se ve un template con los tags para insertar información dinámica proveniente de los modelos WebML.

3.2.2 Modelado con WebRatio

Durante la etapa de toma de requisitos la navegación se capturó mediante diagramas UID, sumados a los mockups HTML y tests de interacción. Cada una de estas partes de la captura de requerimientos ayuda a construir los correspondientes modelos, como se muestra a continuación.

Modelo de Datos

A partir de los diagramas UID podemos comenzar a extraer las entidades principales de la aplicación. Durante esta etapa se crea un modelo de Entidad-Relación tradicional, que luego se sincronizará automáticamente con una base de datos relacional. Si bien presenta la gran ventaja de ser un modelo ampliamente conocido y utilizado, como modelo central de la aplicación no es muy rico, dado que toda la lógica de la misma se ve trasladada al modelo de navegación que termina concentrando todo posible comportamiento ligado no sólo a la navegación, sino a cualquier aspecto de dicha aplicación. Por ejemplo, un procesamiento que se realiza exclusivamente del lado del servidor sin intervención de un usuario, como puede ser el envío automático de una notificación por email, no tiene ningún aspecto navegacional, sin embargo el único lugar donde se puede representar es en el posterior diagrama WebML.

Un detalle adicional respecto del modelo de datos que ofrece WebRatio está en que se pueden crear entidades **volátiles**. Dado que toda la información que se muestra en el diagrama de navegación surge de las entidades representadas en el modelo de datos, y considerando que no siempre esta información se persiste, WebRatio permite la posibilidad de crear entidades que sólo permanecerán en memoria. Por ejemplo, un carrito de compras podría representarse de esta manera, en el caso en que sólo quisiéramos mantener los productos mientras dure la sesión del usuario y no persistirlos más allá de ello.

En nuestra aplicación de ejemplo, podemos obtener las primeras entidades fácilmente viendo los UIDs, que modelan el intercambio de información. Así detectamos las entidades **User**, **Album** y **Photo** junto con sus atributos y relaciones principales, como se ve en el modelo de la figura 3.9.

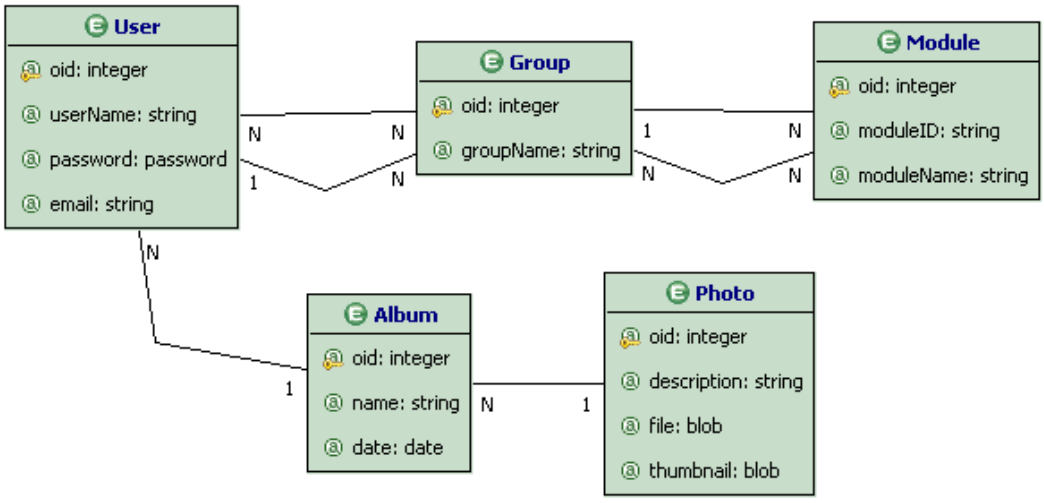


Figura 3.9: Modelo de datos para PhotoShare

El resto de las entidades que se ven en el diagrama, Group y Module, dan marco a un sistema de usuarios basado en grupos y módulos de acceso restringido (en WebRatio un módulo puede ser una página, un conjunto de páginas, una unidad o una vista, por ejemplo). Es un esquema que propone WebRatio para toda aplicación recién creada para

dar soporte a un sistema de usuarios básico, y generalmente resulta de utilidad para cualquier aplicación que requiera de este manejo.

Modelo de Navegación

El modelo más característico, y donde sucede la mayoría del trabajo de diseño es sin dudas el de navegación. Para esto, WebRatio soporta el lenguaje WebML. Con este lenguaje podemos representar páginas, formularios, links, listas, y todo lo que una aplicación web necesita, incluyendo funcionalidad para aplicaciones RIA, como soporte para Ajax. El lenguaje WebML soportado por WebRatio consiste de diferentes unidades ("units") que representan elementos visuales, de navegación o de manipulación del modelo de datos. Algunas de estas unidades están explicadas en la tabla 3.10.

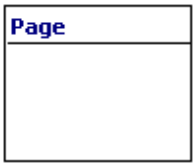
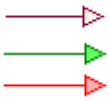
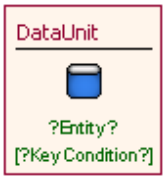
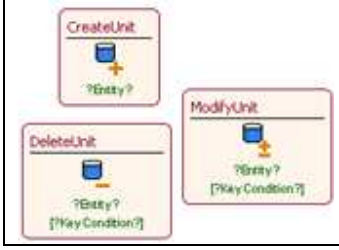
Unidad	Detalle
	**Page:** representa una **página**. Básicamente se trata de un contenedor, y se corresponderá con una url específica. A partir de ella puede partir links.
	**Links:** existen tres tipos de vínculos. Por un lado están los normales, que simplemente conducen de una unidad a otra (o acarrean datos, si son *de transporte*). También están los links **OK** y **KO** (verde y rojo en la imagen), que se usan para conducir la navegación desde una acción exitosa y una fallida, respectivamente.
	**IndexUnit:** se liga a una entidad del modelo de datos y genera una lista de elementos (o entidades concretas). Puede aplicársele un filtro condicional para listar sólo algunos elementos.
	**EntryUnit:** representa un formulario. Típicamente conducirá a una unidad que provoque un alta, baja o modificación en el modelo de datos.
	**DataUnit:** se utiliza para extraer información ligada a una entidad concreta del modelo de datos, y mostrarla por ejemplo en una **page**.
	**CreateUnit / ModifyUnit / DeleteUnit**: estas tres unidades son las que acceden y modifican el modelo de datos. También hay unidades específicas para las relaciones bidireccionales y unidades **script** para ejecutar consultas directamente sobre la base de datos.

Tabla 3.10: Unidades WebML soportadas por WebRatio.

Durante la etapa de diseño del modelo de navegación, podemos basarnos en los artefactos de requerimientos como los UIDs, pero lo que va a guiar nuestro proceso paso a paso serán los tests de interacción generados en el paso anterior para un conjunto acotado de requerimientos. Nuestra tarea estará completa una vez que todos esos tests pasen, y dado que se trata de tests de interacción, la manera de implementar la lógica necesaria para satisfacer la funcionalidad especificada en ellos es modelando la interacción del usuario en el modelo navegacional.

Consideremos por ejemplo en la aplicación PhotoShare, la funcionalidad de cargar fotos en un álbum. Según el UID, un usuario puede primero seleccionar un álbum de su lista de albums, y una vez que ingresa puede ver sus detalles (un nombre y una fecha de creación) junto con las fotos que están cargadas en él, teniendo la opción de eliminar cada una de ellas. Además el usuario puede cargar fotos nuevas seleccionando un archivo y describiendo la foto con un epígrafe. Veamos en la figura 3.11 cómo se puede modelar esta funcionalidad.

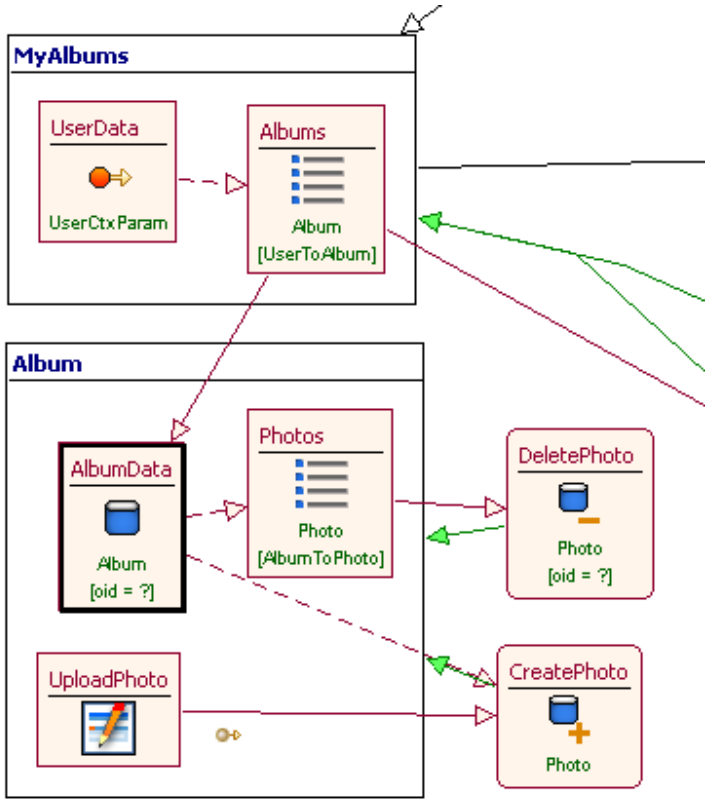


Figura 3.11: Modelo de navegación para PhotoShare

En la porción de modelo que se ve en la figura, podemos notar que hay dos páginas. La página superior, titulara "MyAlbums" es la que contiene el listado de álbumes del usuario, como se puede ver en la IndexUnit de nombre "Albums". La unidad "UserData" simplemente proporciona a la unidad "Albums" la información del usuario que está actualmente navegando el sitio para obtener solamente los álbumes que le pertenecen.

De la unidad que lista los álbumes parte un link hacia la unidad "AlbumData" que se encuentra en la página "Album". Esto indica que cada album del listado de "MyAlbums" poseerá un link que lleve a la unidad que muestre sus detalles. Además, esta unidad ("AlbumData") está en el contexto de una página que contiene dos unidades más:

- La unidad "Photos", que es una IndexUnit que lista todas las fotos del álbum. Similarmente a lo que ocurría entre "UserData" y "Albums", la unidad "AlbumData" le indicará a "Photos" cuáles son las fotos que tiene que seleccionar y que corresponden solamente a ese album. Nótese que de esta unidad parte un link hacia una unidad de borrado, "DeletePhoto", lo que provocará que cada una de las fotos listadas contenga un link para ser eliminada por medio de esta unidad.
- La unidad "UploadPhoto", que es una EntryUnit (lo que representa un formulario) donde se ingresarán los datos de una nueva foto a cargar en el álbum. "UploadPhoto" se comunica con una unidad de alta que agrega la foto a la base de datos.

Modelo de Presentación:

Una vez que la aplicación se comporta como esperamos, podemos enfocarnos en la presentación. Para este modelo tenemos una importante ayuda del lado de los requerimientos: los Mockups HTML, que en la primera etapa del desarrollo sirvieron para establecer cómo debía verse la aplicación frente a los clientes, ahora puede convertirse en la presentación misma.

Para darle a la aplicación generada automáticamente por WebRatio el aspecto de los mockups, podemos generar templates HTML, tomando cada parte de los mockups y colocando código dinámico donde es necesario interactuar con el servidor. Por ejemplo, el siguiente código tomado del mockup HTML que representa el listado de fotos:

```
<ul id="photos-list">
      <li>
         <a href="#">
            <img src="photos/sample-1.jpg" title="test 1"/>
         </a>
      </li>

      <li>
         <a href="#">
            <img src="photos/sample-2.jpg" title="test 2"/>
         </a>
      </li>
      ...
</ul>
```

puede ser traducido a un template para el listado de la unidad "Photos" del diagrama WebML mostrado anteriormente, de la siguiente manera:

```
<ul id="photos-list">
<c:forEach var="current" varStatus="status"
items="${<wr:UnitId/>.data}">
   <c:set var="index" value="${status.index}"/>
   [%def path=unit.selectSingleNode("layout:Attribute[1]")%]
   [%def title=unit.selectSingleNode("layout:Attribute[2]")%]
   <li>
      <a href="<wr:URL context="linkToPhoto"/>">
         <img
            src="<wr:Value context="path"/>"
            title="<wr:Value context="title"/>"
            alt="<wr:Value context="title"/>"
         />
         </a>
      </li>
</ul>
```

Como se ve en el código anterior, se toma del mockups la lista HTML de ejemplo y se agrega código para generarla dinámicamente, iterando entre todas las imágenes de la unidad "Photos". El iterador <c:forEach> recorre cada elemento de dicha unidad, y por cada uno de ellos extrae la ruta de la imagen y la descripción. Luego en la lista se utiliza la ruta para la propiedad **src** de un tag **img**, y la descripción tanto como texto alternativo como para título.

Además se pueden modularizar porciones de la presentación (como por ejemplo un encabezado, pie de página o menú), como permite cualquier motor de templates conocido.

3.3 Evolución y cambios en los requerimientos

Habiendo completado una iteración entera, y verificando que todos los tests para la funcionalidad resuelta pasan, ya podemos agregar un nuevo requerimiento para continuar el ciclo. Como se explicó anteriormente, la idea es ir atacando los requerimientos de a uno o en pequeños grupos relacionados por funcionalidad, de manera que se pueda ir avanzando de la misma manera que en TDD tradicional.

Para ejemplificar el agregado de un nuevo requerimiento, tomemos la funcionalidad de agregar amigos de nuestra aplicación PhotoShare. Para esto, tenemos que volver a seguir los pasos de la metodología aplicados anteriormente:

1. Modelar los nuevos requerimientos usando UIDs, y opcionalmente documentación tradicional como Use Cases o User Stories.
2. Crear nuevos mockups de presentación para la funcionalidad agregada, o modificar los existentes.

3. Generar tests de interacción para validar la nueva funcionalidad sobre los mockups creados o alterados en el paso previo. Estos tests deberían fallar debido a que el desarrollo aún no fue llevado a cabo.
4. Modelar la nueva funcionalidad teniendo en cuenta la interacción capturada en los UIDs y los requisitos de los tests.
5. Correr los tests y comprobar que pasan. Si algún test falla, volver al paso anterior para corregir los errores hasta que los tests finalmente pasen.

Veremos en primer lugar las User Stories asociadas a la nueva funcionalidad de buscar y agregar amigos, que nos ayudarán en los siguientes pasos de la etapa de toma de requerimientos:

1. Como usuario registrado deseo realizar una búsqueda de personas para agregar a mis contactos (search person)

- el usuario ingresa uno o mas términos de búsqueda
- el sistema muestra una lista de contactos que coinciden con los parámetros de búsqueda ingresados
- solo se mostrarán personas que no sean actualmente contacto del usuario
- para cada resultado de búsqueda se muestra la foto y nombre completo de la persona

2. Como usuario registrado deseo agregar una persona a mis contactos (invite person to be in contact)

- el usuario envía una invitación a la persona
- la invitación queda pendiente hasta que la persona en cuestión acepte o rechace la invitación

3. Como usuario registrado deseo cancelar una invitación de contacto enviada a una persona (cancel invitation)

- el usuario selecciona la opción de cancelar invitación para un contacto
- la persona involucrada en la invitación ya no podrá aceptar la invitación ya que la misma deja de estar activa

4. Como usuario registrado deseo aceptar una invitación para ser contacto de la persona que la envió (accept invitation)

- la persona acepta una invitación pendiente
- el contacto se agrega a la lista de contactos
- el usuario puede acceder al sitio del contacto

5. Como usuario registrado deseo rechazar una invitación de contacto con lo cual la persona no será uno de mis contactos (reject invitation)

Para satisfacer estos requisitos debemos generar diagramas UID que indiquen cómo será el intercambio de información especificado. Para ejemplificar, veamos cómo sería un diagrama para el caso de la historia número 2, en la que un usuario desea agregar a otro a su lista de contactos. Asumimos que anteriormente se realizó una búsqueda de personas y que estamos frente a una lista de posibles contactos.

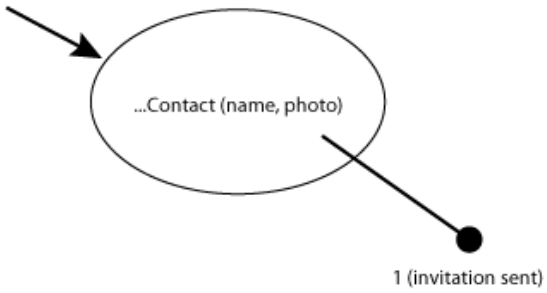


Figura 3.12: Diagrama UID para el agregado de contactos en PhotoShare

La figura 3.12 simplemente muestra cómo a partir de una lista de usuarios se selecciona uno para enviar la invitación. A nivel de mockups, se debería mostrar la presentación de el listado de contactos y alguna manera de enviar una invitación a cualquiera de ellos. Generamos entonces un mockup para terminar de detallar cómo se verá esta funcionalidad en términos generales, y cómo se realizará la interacción de enviar una invitación.

Figura 3.13: Mockup de extensión para PhotoShare.

En la figura 3.13 podemos ver un listado de usuarios, y junto a cada uno de ellos un link que permite enviar una invitación. En esta etapa deberíamos generar también mockups para todos los caminos de interacción descriptos en las historias de usuario, como por ejemplo el anuncio que indica que la invitación fue enviada, aceptada o rechazada.

Una vez que generamos todos los UIDs y los mockups HTML asociados, deberíamos especificar más formalmente la interacción a través de tests de interacción. Veamos qué se necesita para verificar que la funcionalidad de enviar una invitación funciona correctamente, suponiendo que tenemos dos usuarios creados en el sistema, llamados "Butch Trucks" y "Duane Allman", y que el primero quiere agregar al segundo a su lista de contactos, enviándole una invitación. Deberíamos entonces verificar que si "Butch Trucks" envía una invitación a "Duane Allman", al ingresar el segundo a PhotoShare debería ver el anuncio. En detalle:

1. Deberíamos ingresar a PhotoShare con el usuario btrucks y su contraseña correcta.
2. Ingresar el término "Duane Allman" en el campo "search" para buscar personas en PhotoShare.
3. En el listado de personas, seleccionar el vínculo "invite" que se encuentra al lado de "Duane Allman".
4. Salir de PhotoShare
5. Ingresar a PhotoShare con el usuario dallman y su contraseña correcta.
6. Debería aparecer el texto "1 pending invitation from Butch Trucks".

Al generar este test con Selenium y correrlo, el mismo fallará, dado que la funcionalidad no está aún implementada.

Para ilustrar la implementación del agregado de contactos en PhotoShare, vamos a comenzar por el modelo de datos. Lo que necesitamos respecto a este modelo es, por un lado agregar la posibilidad a los usuarios (de la entidad **User**) de conocer a otros usuarios, y por otro la representación de las invitaciones, para que antes de establecer una relación entre dos usuarios se registre la invitación relacionada tanto con el usuario que la solicita como con el que la recibe. Esta invitación luego será eliminada por el usuario receptor tanto si la acepta como si la rechaza, la diferencia estará en que al aceptarla establecerá además el vínculo entre los dos usuarios.

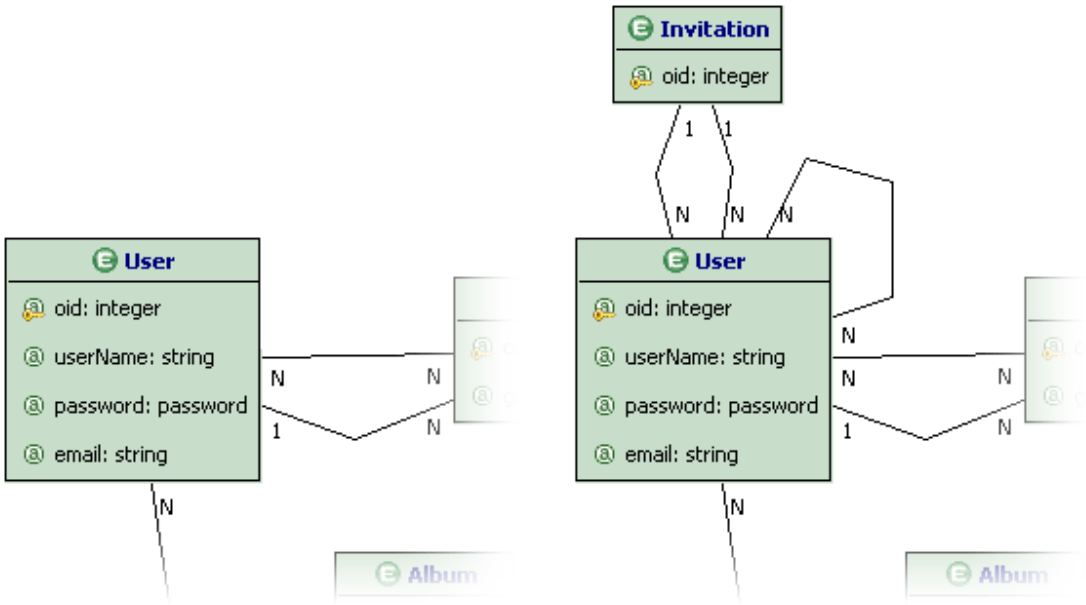


Figura 3.14: Extensión del modelo de datos de PhotoShare.

En la figura 3.14 se muestra el agregado de la relación recursiva N a N desde y hacia la entidad **User**, y además el agregado de la entidad **Invitation** con dos relaciones, una 1 a N con el usuario que envía la invitación y otra también 1 a N con el usuario que la recibe.

Una vez listo el modelo de datos, pasamos a extender el modelo de navegación para soportar la administración de contactos. En este punto hay varias cosas para agregar: por un lado necesitamos que en el momento de ingresar a la aplicación todo usuario vea la lista de invitaciones disponibles, tal como se muestra en el mockup correspondiente. Por otra parte también tenemos que implementar la funcionalidad de invitar, aceptar y rechazar una invitación.

En la figura 3.15 vemos cómo se agrega el listado de invitaciones pendientes a la página principal:

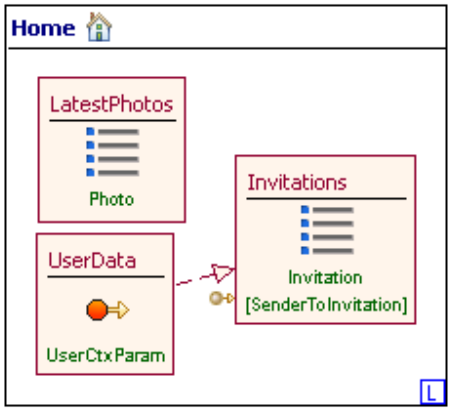


Figura 3.15: Extensión de PhotoShare al modelo WebML.

Sumado a la unidad LatestPhotos, que agregamos previamente para mostrar las últimas fotos cargadas, agregamos la unidad Invitations (de tipo indexUnit, que se emplea para listar elementos) muestra una lista de invitaciones. La unidad UserData simplemente indica a Invitations que sólo tiene que mostrar las invitaciones enviadas al usuario que está actualmente conectado al sitio.

Siguiendo con las funcionalidades necesarias para pasar los nuevos tests, tenemos que implementar la posibilidad de enviar invitaciones a otros usuarios. En la página destinada a buscar personas, tendríamos que agregar al listado de resultados una forma de enviar invitaciones a cada uno de ellos, si es que no forman ya parte de la lista de contactos, o no tienen una invitación pendiente de parte del usuario conectado. Agregamos entonces un vínculo a la unidad de los resultados.

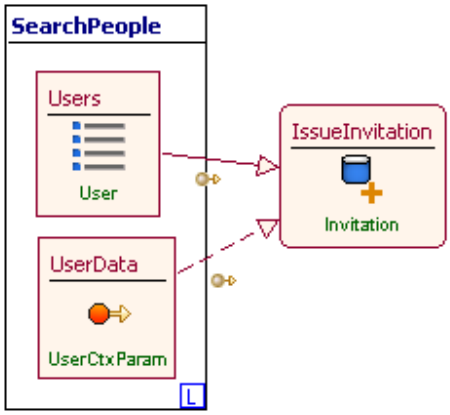


Figura 3.16: Modelado de invitaciones de PhotoShare en WebML.

Siguiendo con las funcionalidades necesarias para pasar los nuevos tests, tenemos que implementar la posibilidad de enviar invitaciones a otros usuarios. En la página que muestra los resultados de búsqueda de personas (unidad **Users** de la figura 3.16) agregamos un vínculo a la unidad **IssueInvitation** que crea una invitación nueva, con el usuario del listado como destinatario y el usuario actualmente conectado al sitio como remitente. El vínculo se llamará "invite" como indica el test detallado al principio de esta sección.

La siguiente tarea consiste en adaptar la presentación de la nueva funcionalidad conforme a los mockups creados durante la captura de requerimientos. Al crear los nuevos templates de presentación, los tests se deberían poder correr sin problemas, ahora sobre el prototipo, pudiendo esperar errores en las aserciones pero no fallos que impidan su ejecución. No obstante, puede ser necesario que los tests deban adaptarse a la aplicación, que puede presentar algunas diferencias respecto de los mockups, por ejemplo si existe alguna imposición de la tecnología sobre los tags HTML, especialmente en los puntos donde la presentación se genera automáticamente. Otra diferencia que seguramente se da en todos los casos es la diferencia entre las URLs de los mockups, que hacen referencia a archivos estáticos (como por ejemplo "file:///D:/Documents/PhotoShare/Mockups /index.html") contra las URLs reales de la aplicación dinámica.

Una vez readaptados los tests de interacción, podemos correrlos sobre el prototipo generado para comprobar que la funcionalidad implementada cumple con lo esperado en los requerimientos.

Si los tests fallan, entonces volveremos a buscar los errores en los modelos, corregirlos y volver a correr los mismos tests hasta que pasen. En este punto también se deben correr todos los tests acumulados hasta el momento para comprobar que al implementar nuevas funcionalidades no introdujimos errores nuevos a las existentes, de la misma forma que se comprueba en TDD que un refactoring de código no compromete el funcionamiento de la aplicación.

Cuando todos los tests pasan podemos dar por concluido el ciclo y tomar nuevos requerimientos para continuar con el desarrollo de la aplicación.

Capítulo 4
Experiencia: comparación con metodologías tradicionales

Para poder comprobar las presuntas ventajas que ofrece la metodología explicada en este trabajo, se realizó una experiencia de desarrollo en el marco de la materia Diseño de Aplicaciones en la Web de la Facultad de Informática de la UNLP. Esta materia optativa para alumnos avanzados de la carrera cubre diferentes temas relacionados al estado del arte del desarrollo de aplicaciones web.

Como trabajo práctico de la materia se propuso a los estudiantes que realizaran una aplicación simplificada de microblogging similar a Twitter[8], en la que se cubrían varios casos de uso relacionados por ejemplo con la creación de posts, seguimiento y búsqueda de usuarios o anuncios automatizados. Los alumnos desarrollaron completamente estos casos, presentación inclusive, y pusieron a prueba sus desarrollos mediante tests de interacción otorgados por la cátedra. La experiencia constó de dos etapas durante las cuales los distintos grupos de alumnos realizaron dos desarrollos empleando diferentes metodologías, una de las cuales constó de una síntesis de la metodología aquí propuesta.

A continuación se detallará el enunciado entregado a los alumnos, luego la manera de abordar el experimento, y finalmente los resultados obtenidos junto con una breve conclusión.

[8] http://www.twitter.com/

4.1 Trabajo solicitado

El enunciado del trabajo que se dio a los alumnos concretamente fue el siguiente:

"Se desea modelar, diseñar e implementar una aplicación web de Microblogging similar a Twitter. La aplicación deberá permitir a los usuarios publicar mensajes breves (aproximadamente 150 caracteres) que serán publicados en el listado de publicaciones de la página correspondiente al Usuario.

Cada nuevo usuario deberá estar registrado para poder utilizar el servicio y para ello la aplicación deberá proveer la funcionalidad necesaria para el proceso de registración de un usuario. Sin embargo, cualquier usuario anónimo (no logueado) podrá visitar las páginas de los usuarios registrados para leer las publicaciones.

La aplicación deberá proveer soporte tanto para HashTags (soporte para tags) como para Microblog (referencias a otros usuarios)."

La manera de desarrollar se planteó en dos opciones diferentes. Por un lado, la opción de implementar con un framework tradicional, como Struts[9] o Groovy on Rails[10], por el otro lado la implementación empleando modelos WebML a través de WebRatio, la misma herramienta presentada en las secciones previas para ejemplificar el desarrollo

4.2 Planeamiento del experimento

Para organizar el desarrollo y obtener una buena muestra de resultados, se dividió al alumnado en equipos de dos personas. A su vez, de la población general de equipos, se la dividió en dos grandes grupos, a los cuales llamaremos A y B.

En una primera etapa, todos los equipos desarrollaron la aplicación completa utilizando WebRatio, la diferencia estuvo en que los equipos del grupo A se basaron en la técnica de TDD, mientras que el grupo B desarrollaron libremente.

En la segunda etapa del cuatrimestre, manteniendo la misma aplicación, los dos grupos realizaron el trabajo pero esta vez cada equipo eligió un framework conocido para desarrollar. Nuevamente, la diferencia entre los equipos del grupo A y los del grupo B estuvo en la forma de abordar el desarrollo: en este caso el grupo B fue el que aplicó TDD usando tests de interacción, mientras que el grupo A desarrolló de manera tradicional, es decir sin usar tests o usándolos al final del desarrollo.

[9] http://struts.apache.org/
[10] http://www.grails.org/

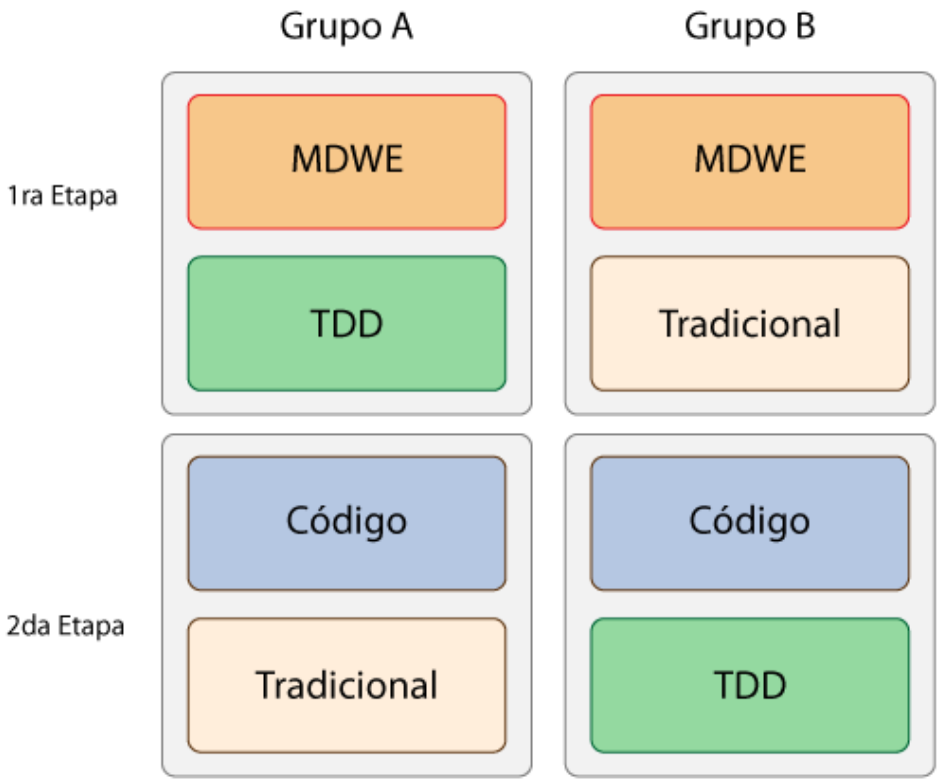


Figura 4.1: Plan del experimento.

Cabe destacar que este plan no sólo sirvió para comparar la metodología propuesta con una metodología enteramente tradicional, sino que gracias a las combinaciones que se dieron también se pudo comparar el efecto de usar tests de interacción dentro de un esquema tradicional de desarrollo, y por otra parte también se pudo apreciar la diferencia entre emplear una metodología basada en modelos y una basada en escritura de código bajo las reglas de TDD. Surgió también la posibilidad de evaluar si la metodología propuesta en este trabajo podría funcionar de igual manera sin usar exclusivamente MDWE durante la etapa de desarrollo, aunque aprovechando de todas maneras las otras componentes de la propuesta en lo referido a la captura de requerimientos y forma de abordar los mismos en etapas cortas. Sin embargo, el experimento no alcanzó para poner a prueba esta posibilidad de forma específica, dado que fue detectada demasiado tarde, ya iniciado el mismo.

En todos los desarrollos se pidió a cada equipo que tomara el tiempo de desarrollo para obtener una medida. Este tiempo no debe ser tomado fuera del contexto de cada equipo, dado que en el caso del uso de WebRatio existe una carga adicional de horas dedicadas al aprendizaje de la herramienta, que ningún alumno había usado anteriormente.

4.3 Resultados

Los resultados evaluados en primera instancia fueron las medidas de tiempo. Todos los grupos habían informado estimativamente la cantidad de horas que había llevado el desarrollo en las diferentes iteraciones, en algunos casos discriminando el porcentaje invertido en aprender la tecnología, implementar, correr los tests, y todas las actividades involucradas.

La medida de horas, contraponiendo los equipos que desarrollaron empleando la metodología propuesta (en adelante "TDD" para simplificar) vs. los que desarrollaron de manera tradicional no resultó demasiado favorable. En muchos casos los tiempos eran mayores o bajaban poco, teniendo en cuenta que la experiencia de la primera etapa tendría que haber facilitado el trabajo, sobre todo en el caso del grupo B donde el desarrollo con TDD se realizaba en la segunda etapa.

Observando mejor los tiempos se concluyó que los resultados podían ser entendibles, dado que el trabajar constantemente contra las especificaciones rígidas de los tests de aceptación podía conllevar algo más de tiempo en cada iteración, hasta ver que tales tests estaban satisfechos. Las medidas de tiempo para los grupos A y B pueden observarse en los gráficos 4.2 y 4.3 respectivamente.

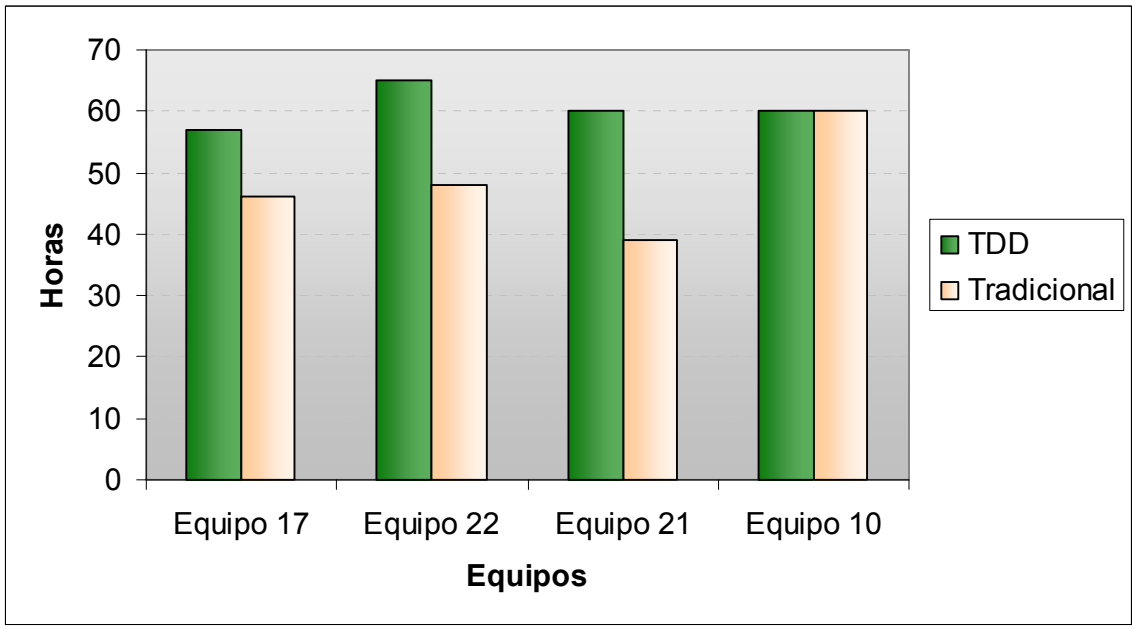


Figura 4.2: Tiempos de desarrollo en horas del grupo A.

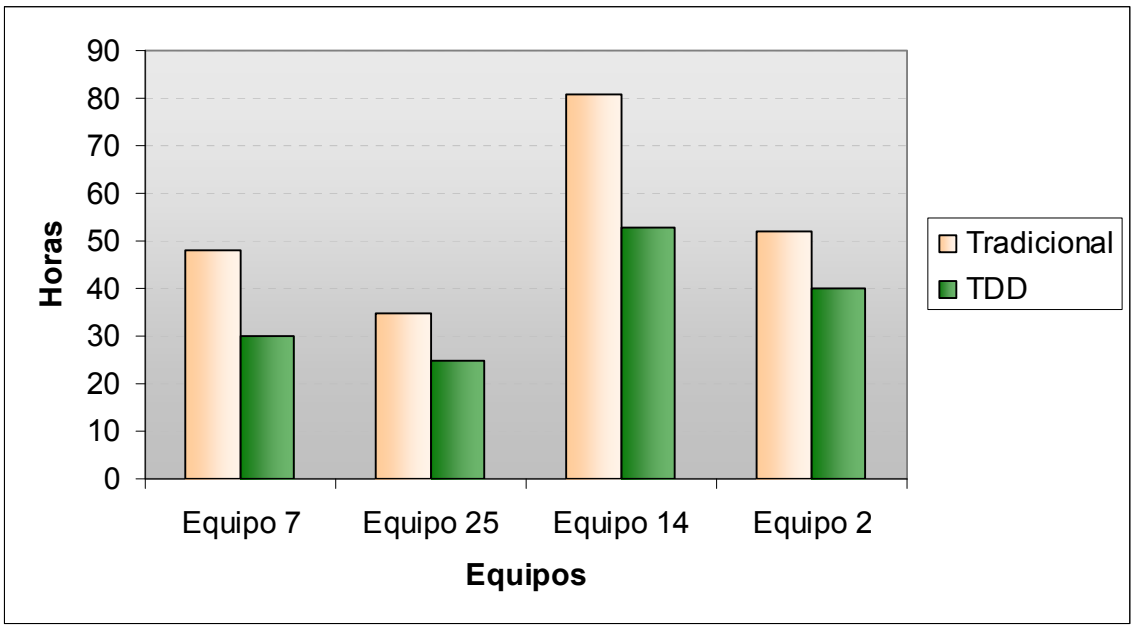


Figura 4.3: Tiempos de desarrollo en horas del grupo B.

Al ver los gráficos de ambos grupos, es importante también tener en cuenta que la primera etapa, que se corresponde con la primera barra de cada equipo, indica que el desarrollo fue hecho usando herramientas dirigidas por modelos mientras que la segunda barra indica un desarrollo hecho con una metodología de escritura manual de código usando diferentes frameworks conocidos.

Como se mencionó antes, el hecho de que en estos resultados los desarrollos realizados con la metodología propuesta hayan llevado relativamente más tiempo que lo desarrollados usando metodologías tradicionales puede mitigarse si consideráramos que estar desarrollando constantemente contra un conjunto de tests siempre resulta más exigente en todo momento. Sin embargo, también es esperable que el tiempo invertido se traduzca en mejor calidad del producto final, en el cual todos los tests pasan. Podemos pasar entonces a otra medida interesante, que consiste en observar la cantidad de tests pasados por cada etapa, que se puede ver en las tablas 4.4 y 4.5 para los grupos A y B respectivamente.

	TDD	Tradicional
Equipo 17	33/36	36/36
Equipo 22	36/36	35/36
Equipo 21	36/36	35/36
Equipo 10	30/36	36/36

Tabla 4.4: Tests pasados por el grupo A.

	Tradicional	TDD
Equipo 7	30/36	12/36
Equipo 25	10/36	36/36
Equipo 14	0/36	0/36
Equipo 2	0/36	25/36

Tabla 4.5: Tests pasados por el grupo B.

Desde un punto de vista, los resultados, si bien son bastante positivos, no terminan de ser concluyentes. Podemos ver una mejoría considerable en casi todos los casos, pero sólo se ven diferencias claras en los equipos 25 y 2 del grupo B. En el grupo A, en cambio, se ve que hay más tests que pasan en el desarrollo tradicional que en TDD, lo cual también es entendible, dado que la segunda etapa siempre tiene el sesgo de contar con la experiencia del desarrollo de la primera.

Consideremos no obstante un segundo punto de vista para los resultados de cantidad de tests pasados. Si comparamos la cantidad de tests del grupo A en la primera etapa, contra los del grupo B en la misma etapa donde ambos estaban en igualdad de condiciones (o incluso con la ventaja del grupo B de enfrentar un desarrollo con una metodología conocida) podemos ver una mejora mucho más contundente: la sumatoria de tests pasados en el grupo A es de **135/144**, contra apenas **40/144** del grupo B. Esta nueva evidencia resulta mucho más favorable para la metodología del estilo TDD.

Hay una última medida tomada para terminar de analizar las metodologías, y tiene que ver con el porcentaje de funcionalidad cubierto en cada etapa de la implementación. Este porcentaje es obtenido comparando los requisitos contra tests de aceptación manuales, esto es, probando la aplicación punto por punto manualmente para comprobar que todo funcione. De esta manera, si bien estamos yendo en contra de la idea de usar tests de interacción automatizados, evitamos algunos problemas técnicos por los cuales los tests de interacción puedan estar fallando, dando falsos negativos. Por ejemplo en uno de los informes observamos la siguiente declaración:

"Todos los requerimientos fueron satisfechos y los tests pasaron exitosamente excepto dos tests del SearchTestCase por el siguiente motivo: no nos gustó que el test luego de borrar un tweet realice la búsqueda con el texto del tweet borrado y no espera encontrar el string dentro de la totalidad de la página, pero en nuestra implementación mostramos un cartel indicando que la búsqueda con el texto X no ha tenido resultados, así lo hace Twitter, es una decisión que defendemos."

Este tipo de situaciones nos llevó a pensar que los tests manuales podían ofrecer algunas mejoras en la calidad de la medida para lo que respecta a este experimento. Veamos entonces las medidas de porcentaje de funcionalidad cubierto para ambos grupos, en los gráficos 4.6 y 4.7 para los grupos A y B respectivamente.

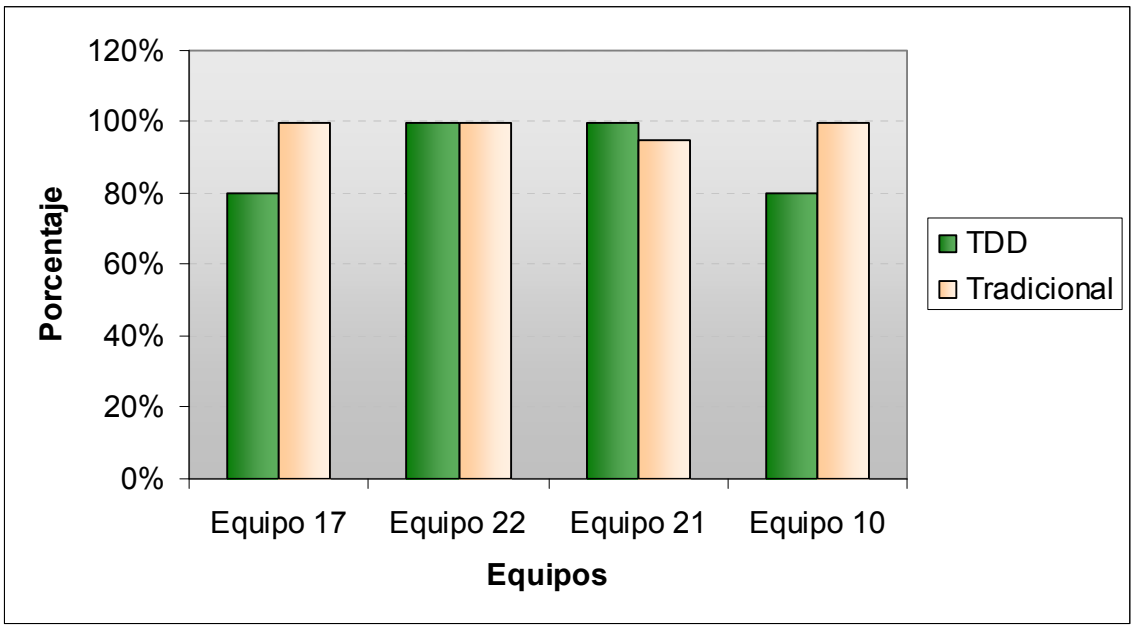


Figura 4.6: Porcentajes de funcionalidad cubiertos por el grupo A.

Tomemos primero el caso del grupo A, en el gráfico 4.6. Como se puede observar, la mejora esperada en la segunda etapa favorece a la metodología tradicional. Si vemos en cambio el gráfico del grupo B, observamos también que la segunda barra está en casi todos los casos más alta, cubriendo un mayor porcentaje de la funcionalidad total.

A las medidas que vemos en ambos gráficos deberíamos quitarles el sesgo que da la facilidad de haber pasado ya por una etapa completa de implementación de la misma aplicación, pero una vez más la medida interesante no está entre dos etapas del mismo grupo, sino en la comparación entre ambos grupos. En este sentido resulta destacable la diferencia que hay entre el grupo A y el grupo B, si consideramos el porcentaje cubierto en la primera etapa. En el grupo A se ve un promedio de **90%** cubierto en la primera etapa, contra apenas un **63.75%** del grupo B.

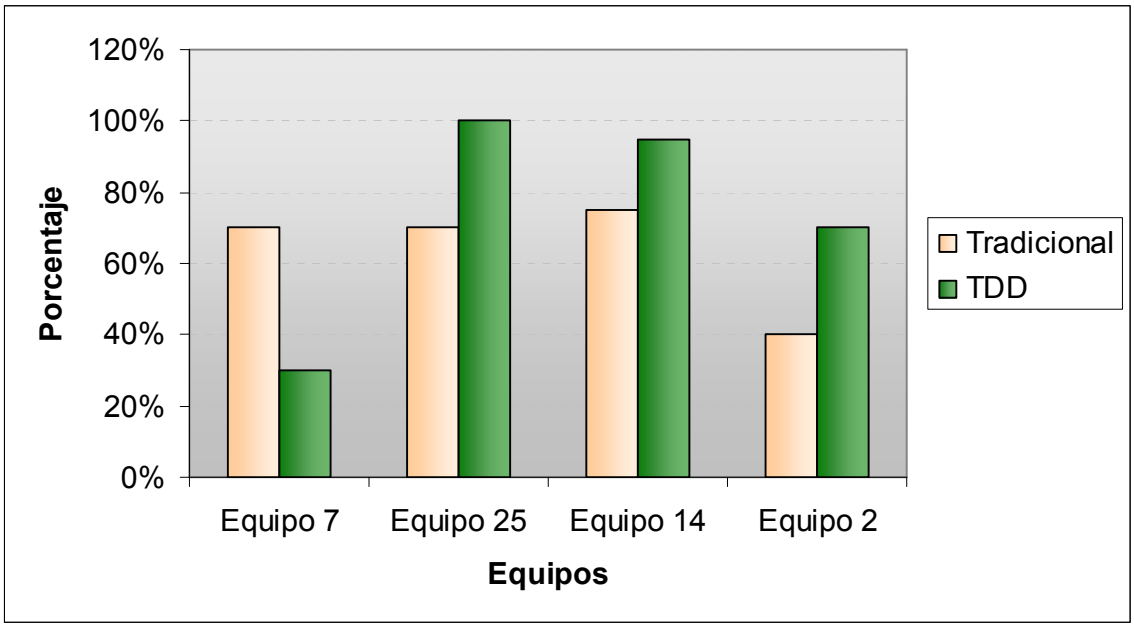


Figura 4.7: Porcentajes de funcionalidad cubiertos por el grupo A.

La validez de los resultados obtenidos merece un análisis aparte. Si bien las medidas tomadas tienen una cierta coherencia respecto de la expectativa, hay que tener en cuenta que existen varias condiciones que pueden haber comprometido el valor de las mismas, a saber:

- La población es demasiado pequeña: con solamente ocho grupos de alumnos, cualquier varianza influye notablemente en los resultados. Por ejemplo, los grupos que pasaron muy pocos tests pueden haber tenido problemas técnicos no relacionados directamente con el funcionamiento de la aplicación, sino la especificación de los tests, y sin embargo se obtiene una medida extremadamente baja que en una muestra de cuatro valores compromete el promedio.
- La medida de tests pasados no hace justicia a los grupos que no usaron TDD en la primera etapa: si bien los tests fueron adaptados, los grupos no tenían la posibilidad de correrlos mientras programaban, por lo cual están en desventaja. Si bien es una de las condiciones que nos interesaba medir, tendría que haber una compensación en este sentido. De nuevo, los tests de aceptación manuales pasan por alto estos temas y ayudan a mantener la veracidad de la cobertura en la funcionalidad.
- La aplicación es la misma en ambas etapas: claramente, en la segunda etapa todo se facilita mucho más. Intentamos contrarrestar este efecto agregando nuevos requerimientos complejos para que el desarrollo no sea exactamente el mismo. Además al cambiar la herramienta de desarrollo (Model-Driven en la primera etapa, escritura de código en la segunda) se buscó que los alumnos no

puedan reutilizar lo programado en la primera. No obstante a ello, el sesgo de repetir los requerimientos es considerable y se notó en los resultados.

- Los sujetos no siempre trabajaban en la industria: como se explicó al principio de este capítulo, los desarrolladores eran alumnos de una materia de grado, con lo cual muchos no tenían experiencia profesional y esto puede haber traído problemas en la medición de horas y en la manera de desarrollar.

4.4 Conclusiones

Las conclusiones del experimento favorecen a la metodología propuesta, pero en aspectos que no son precisamente las que se esperaban al momento de diseñar el experimento. Al enfrentar el desarrollo con diferentes tecnologías y diferentes maneras de usar los tests de interacción, esperábamos tener mejoras en los tiempos de desarrollo como meta principal, pero los resultados arrojaron otros beneficios que no estábamos considerando en primera instancia.

Los tiempos de desarrollo, como se pudo ver en la sección anterior, no mejoraron notablemente respecto de las metodologías que no usan tests, no obstante hubo mejoras substanciales en la calidad de la aplicación obtenida, ya sea en términos de tests pasados como en cobertura total de la funcionalidad implementada medida a través de tests de aceptación.

Otra ventaja que los resultados no muestran, pero que se notó en los alumnos que participaron del experimento es la diferencia de motivación: una ventaja que reclama tener la técnica de TDD tradicional es la de mantener motivados a los desarrolladores y con objetivos firmes . Esto mismo se notó claramente en todos los grupos, incluso los que no fueron informados en esta sección. Todos los que desarrollaron empleando la metodología propuesta en este trabajo pusieron más empeño en programar toda la funcionalidad, porque al tener tests para comprobar que cada pieza de la aplicación funciona como esperaba la cátedra, contaban con más seguridad de que el trabajo estaba bien hecho. Al programar sin tener tests, o corriendo los tests únicamente al final del desarrollo, el nivel de incertidumbre sube.

Otra observación que hicimos fue que los equipos del grupo A, que desarrollaron primero usando TDD y modelos, supieron trasladar mucho mejor la experiencia a la segunda etapa que los del grupo B, quienes desarrollaron también con modelos durante la primera etapa, pero sin la guía de los tests de interacción.

Capítulo 5
Trabajo relacionado

La idea de explotar las ventajas de las metodologías ágiles para desarrollar aplicaciones web ya ha sido explorada anteriormente por diferentes autores. En este capítulo veremos algunos registros de aplicación de estas metodologías junto con otros trabajos relacionados a la metodología propuesta, como el uso de herramientas dirigidas por modelos en un estilo ágil o el uso de tests en diferentes etapas del desarrollo de aplicaciones web.

Particularmente también se detallará en dos ideas bastante novedosas, también con relación a este trabajo, llamadas CubicTest (una herramienta que facilita el uso de Test First Programming para web) y Behavior Driven Development, que consiste en escribir especificaciones del comportamiento de una aplicación antes de desarrollarla, generando tests automáticamente a partir de ellas.

También como trabajo relacionado, cabe destacar que las ideas preliminares de este trabajo fueron presentadas en una publicación junto a otros autores (Robles Luna, Grigera, & Rossi, 2009).

5.1 Trabajos relacionados varios

En lo que se refiere a los beneficios que las metodologías ágiles presentan respecto a la captura y validación de requerimientos junto a los clientes, es importante mencionar el proceso de desarrollo AWE (Agile Web Enginerring) (McDonald & Welland, 2003). En este trabajo, los autores presentan una metodología ágil aplicada a la ingeniería de aplicaciones web centrándose en la estructura de los equipos de desarrollo y la

comunicación entre ellos, haciendo hincapié en la participación de los clientes en todas las etapas, de una manera similar a la que se plantea en la metodología ágil conocida como Extreme Programming (Jeffries, Anderson, & Hendrickson, 2000). Al igual que el presente trabajo, AWE intenta poner a favor del desarrollo web las ventajas de las metodologías ágiles, aunque enfocándose específicamente en la composición e interacción de los diferentes equipos multidisciplinarios que participan del desarrollo, siendo éste un aspecto poco explorado en la metodología aquí propuesta. Sin embargo, en contraste con lo que se propone en el presente trabajo, AWE es solamente un proceso y no indica cómo se obtienen los diferentes artefactos de software a través de él. Además, se plantea en un contexto de desarrollo tradicional mediante escritura de código, cosa que en este trabajo se reemplaza por desarrollo dirigido por modelos.

En referencia a combinar un estilo de desarrollo ágil con metodologías dirigidas por modelos, muchos procesos de ingeniería web como WebML (Ceri, Fraternali, & Bongio, 2000), UWE (Koch, Knapp, Zhang, & Baumeister, 2005), OOHDM (Rossi & Schwabe, 2008), OOWS (Pastor, Abrahão, & Fons, 2001) y OO-H (Cachero & Koch, n.d.) han manifestado ser compatibles con el desarrollo en iteraciones (en contraste con el método clásico de cascada), pero este es solo un aspecto de las metodologías ágiles, y no se han reportado en la literatura intentos concretos de aplicar una metodología ágil completa con MDWE.

En el plano general del desarrollo de software (no sólo en el desarrollo de aplicaciones web), las metodologías ágiles han sido ampliamente adoptadas, aunque generalmente se toma un enfoque orientado a la escritura de código fuente, en contraste con el uso de modelos y transformaciones. Un punto de vista interesante es el de "extreme non-programming" (Pastor, 2006), que, aprovechando un juego de palabras para confrontar con la técnica de extreme programming, propone el uso de modelos únicamente para el desarrollo, entendiendo que la escritura de código fuente es una práctica que luego de 40 años aún no es capaz de producir software de calidad. Sin embargo, en un trabajo reciente relacionado con TDD (Janzen & Saiedian, 2008), los autores indican que la técnica es apropiada también para la etapa de diseño, y muestran ejemplo en los cuales TDD es utilizado más allá de las metodologías *extremas*.

El uso de Test-Driven Development para aplicaciones interactivas es relativamente nuevo, dada la clara distancia que existe entre la funcionalidad que se puede capturar mediante tests de unidad y la presentación o interacción propia de este tipo de aplicaciones. Al enfocarse los tests de unidad en entidades como clases y métodos, no resultan apropiados para hacer tests sobre interfaces gráficas. No obstante, también ha habido intentos de realizar tests de interacción en este sentido, como en el trabajo de Alles et al. (Alles, Crosby, Harleton, Pattison, & Erickson, 2006), donde los autores presentan una técnica para organizar el desarrollo y la codificación de manera de poder producir aplicaciones con sus interfaces debidamente testeadas a partir de historias de usuario.

Siguiendo con la idea de emplear metodologías ágiles para desarrollar aplicaciones web, el trabajo de Pipka (Pipka, 2009) propone el uso de TDD como metodología de desarrollo, enfocándose en las diferentes componentes de la arquitectura Model View Controller, siempre siguiendo un estilo de escritura manual de código.

También relacionado con este trabajo, y dado que TDD hace uso intensivo de modelos de requerimientos, es importante mencionar que la mayoría de los procesos de ingeniearía web tiene modos automáticos o heurísticas explícitas para derivar contenido y modelos de navegación a partir de documentos de requerimientos. Particularmente en

OOWS, el modelo conceptual puede ser generado a partir de los requerimientos empleando transformaciones ente modelos. En un trabajo anterior basado en OOH y UWE (Cachero & Koch, n.d.), los autores presentan una manera de traducir casos de uso a modelos de navegación, dando especial importancia a los documentos de requerimientos.

5.2 Cubic Test

CubicTest[11] es una herramienta basada en Ecplise[12] que sirve para especificar tests de interacción Selenium gráficamente (como se muestra en la figura 5.1, lo que evita tener que escribirlos e incluso la necesidad de tener código HTML de antemano. Estas características hacen que sea una buena alternativa para implementar Test First Programming.

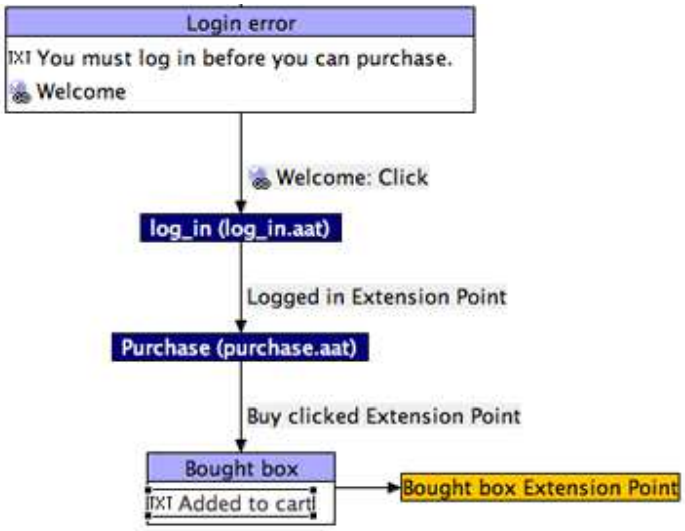


Figura 5.1: Modelo en CubicTest.

CubicTest podría ser una buena herramienta para implementar parte de la metodología propuesta en este trabajo. Una desventaja que presenta está en que metodológicamente no contempla la generación de tests de interacción mediante la técnica de captura, cosa que puede resultar útil, al menos en una primera especificación. Relacionado a esta restricción está el hecho de no requerir mockups HTML; esta característica resulta particularmente ventajosa a la hora de escribir test en forma programática: permite que nos enfoquemos directamente en los tests, o podríamos decir en el comportamiento de la aplicación, antes de especificar nada relacionado con la interfaz. La desventaja está nuevamente en la captura de tests, pero por otra parte, en la metodología propuesta en este trabajo, deshacerse de los GUI mockups no es una alternativa viable porque se perderían todas las ventajas de convenir el *look&feel* de la aplicación con el usuario. Visto de otra manera, si bien es cierto que CubicTest prescinde del código HTML, no relega completamente el modelo de presentación, porque es

[11] http://cubictest.seleniumhq.org/
[12] http://www.eclipse.org/

necesario especificarlo junto con los tests. Sin embargo, una forma de sacar partido de esta especificación puede ser exportarla a HTML (la herramienta lo permite) y trabajar con el cliente sobre esta presentación.

5.3 Behavior-Driven Development

Una opción para captar requerimientos, muy ligada a la metodología de TDD es el desarrollo conducido por comportamiento (Behavior-Driven Development – BDD).

Originalmente, BDD fue concebido como una evolución a partir de las técnicas TDD y Diseño Dirigido por Dominios (Evans, 2003). Esta última técnica se basa en aprovechar las fortalezas del software Orientado a Objetos para simular fielmente el negocio que se quiere modelar, tratando de establecer un lenguaje común entre el personal técnico (los desarrolladores, por ejemplo) y el personal que tiene el conocimiento acerca del dominio. Este lenguaje tiene el nombre de *Lenguaje Ubicuo*.

La idea detrás de BDD consiste en emplear un lenguaje muy específico y concreto para minimizar la falta o disfuncionalidad de la comunicación entre las partes involucradas en el proceso de desarrollo de software. De esta manera, se pueden escribir requerimientos de una manera clara para el experto de dominio, que a su vez sirva para validar o guiar de alguna manera clara el desarrollo que satisfará dichos requerimientos. Por ejemplo, una especificación puede escribirse de la siguiente manera:

```
As a Role
I request a Feature
To gain a Benefit
```

esto significa "bajo un cierto *rol* requiero una *funcionalidad* para obtener un *beneficio*". Como se puede apreciar, este lenguaje es entendible por cualquier persona; sin embargo, la intención de estas descripciones no es sólo la de proveer documentación uniforme, sino que pueden ser procesadas y utilizado, por ejemplo, para generar tests. Veamos un ejemplo concreto de una implementación de BDD para Ruby on Rails denominada Cucumber[13]:

[13] http://cukes.info/

```
1     Feature: Tasks
2     In order to keep track of tasks
3     People should be able to
4     Create a list of tasks
5
6     Scenario: List Tasks
7     Given that I have created a task "task 1"
8     When I go to the tasks page
9     Then I should see "task 1"
```

Las primeras cuatro líneas describen la funcionalidad general:

"Para llevar cuenta de tareas
La gente debería poder
Crear una lista de tareas"

A partir de la línea 6 se describe un escenario concreto:

"Dado que creé la tarea 'tarea 1'
Cuando voy a la página de tareas
Entonces debería ver 'tarea 1'"

Nuevamente, todo lo que se describe aquí es fácil de leer para cualquier persona, pero empleando expresiones regulares para darle significado a la información mencionadas en estas historias, es posible que Cucumber genere automáticamente tests de unidad. Por ejemplo, el texto de la línea 7 puede convertirse en la siguiente expresión:

```
1  Given /^that I have created a task "(.*)"$/ do |desc|
2  Task.create!(:description => desc)
3  end
```

donde la expresión **"(.*)"** en la línea 1 deja lugar a que el mismo escenario pueda ser empleado para verificar tareas que lleven cualquier nombre, no sólo **"task 1"**. La línea 2 emplea sintaxis del lenguaje concreto (en este caso Ruby) para crear el objeto que se describe en la precondición. Análogamente, la línea 9 del código anterior ("Then I should see 'task 1'" / "Entonces debería ver 'tarea 1'") se puede convertir una aserción. A partir de estas extensiones a las historias se generan tests de unidad contra los cuales se realiza el desarrollo, de la misma manera que en TDD. La diferencia está en que los requerimientos quedan documentados con extrema precisión, y en un lenguaje legible para los clientes/expertos de dominio.

La filosofía de Behavior-Driven Development está muy relacionada a la de este trabajo: se intenta acercar a los desarrolladores con los clientes, tratando de especificar requerimientos de manera clara y concisa para ambas partes. No obstante, BDD está orientado al desarrollo tradicional mediante escritura de código fuente, contrastado con la propuesta de este trabajo de emplear desarrollo dirigido por modelos.

Capítulo 6
Conclusiones

El objetivo principal de este trabajo fue el de proponer una nueva manera de desarrollar aplicaciones web basada en metodologías existentes como Test-Driven Development (Desarrollo Dirigido por Tests) y Model-Driven Web Engineering (Ingeniería Web Dirigida por Modelos), donde se ataquen las deficiencias corrientes en la captura de requerimientos para satisfacer tanto a clientes como desarrolladores por un lado. Por otro lado también se buscó facilitar el desarrollo y la evolución de tales aplicaciones.

A continuación se detallará en las mejoras obtenidas en cada uno de los aspectos recién mencionadas, y adicionalmente se explicará en forma breve cómo se resolvió el problema de conciliar TDD con MDWE, metodologías que en una primer análisis resultaron opuestas e incompatibles.

6.1 Mejoras en la captura de requerimientos

Respecto a las ventajas que sugiere esta metodología en lo referente a la captura de requerimientos, se quiso demostrar que empleando los artefactos y procedimientos detallados en las etapas iniciales se notan mejoras en dos aspectos:

- Por un lado la **comunicación** entre las partes interesadas se hace **más clara**: haciendo un contraste con la sola utilización de User Stories o Use Cases, el agregado de los artefactos aquí propuestos en las etapas tempranas de desarrollo presentan una mejoría notable en la mayoría de los casos. Los mockups de presentación y los tests de interacción que muestran concretamente lo que debe suceder con la aplicación a desarrollar no deja

dudas sobre la funcionalidad y apunta el desarrollo hacia un fin bien preciso, que es una de las ventajas reportadas en el uso de TDD (Maximilien & Williams, 2003) (Rasmusson, 2003), una de las técnicas en las que se basa este método de desarrollo.

- Por otra parte, estos artefactos generan **documentación** que si bien puede no tener la riqueza expresiva de la documentación escrita en forma de prosa tienen a su vez dos ventajas fundamentales sobre ella: en primer lugar es una manera **compacta** y **precisa** de describir la funcionalidad esperada, sin dejar lugar a las potenciales ambigüedades de la documentación textual. Por otro lado, esta documentación debe ser **mantenida obligatoriamente**, dado que forma parte activa del desarrollo. Por citar un ejemplo, los tests de interacción deben ser necesariamente actualizados cuando los requerimientos que validan se modifican, dado que la metodología está fuertemente basada en el uso de tests. La documentación en forma de texto es mucho más tediosa de mantener, y el único motivo para hacerlo es el de la sola documentación, a diferencia de los tests o mockups de presentación que son directamente necesarios durante el desarrollo.

En referencia al agregado de artefactos durante el período de descripción de requerimientos, se podría argumentar en su contra que estamos agregando más carga y complicaciones a una etapa que de por sí toma bastante tiempo. Sin embargo, todo lo que se agrega es aprovechable (si no inevitable, como en el caso del diseño la presentación) en el desarrollo posterior. Particularmente, los tests de interacción están muy cerca de ser tests de aceptación, que además tienen la ventaja de estar automatizados y tienen un enorme valor a la hora de alterar o extender la aplicación.

6.2 Mejoras en la evolución

Las aplicaciones web tienen una tendencia a crecer constantemente, y su ciclo de desarrollo permanece abierto de forma indefinida, y si bien en las aplicaciones de escritorio esto también es cierto, en las aplicaciones web se hace mucho más evidente, agregando la componente de que la mayoría de las últimas tienen un público mucho mayor y heterogéneo que las primeras, y además requieren estabilidad permanente en prácticamente todos los casos. Todo esto hace que la evolución no sólo es exigente respecto de su frecuencia y velocidad, sino que requiere que no se ponga en riesgo la integridad del resto de la aplicación que está en constante producción.

Muchos de los componentes de la metodología propuesta tienen por objetivo atacar los problemas que presenta la evolución de aplicaciones web.

En primer lugar, la decisión de utilizar desarrollo conducido por modelos fue tomada con el objeto de optimizar la velocidad en el cambio, a la vez que se reducían los descuidos típicos de la escritura de código. En este sentido se puede decir que las expectativas se cumplieron. Si bien en algunos casos cuesta más encontrar la manera de resolver problemas específicos sólo con modelos, en el momento de aplicarlos los errores detectados son mucho menores que los que se observan escribiendo código. Cabe destacar que durante las pruebas realizadas no es cierto que todo el desarrollo se haya

exclusivamente mediante modelos. En particular, la presentación y cierta funcionalidad RIA del lado del cliente requiere del uso de lenguajes como HTML, javascript y Groovy.

Por otra parte, y tal vez este sea el punto más importante, la generación de tests de antemano como se hace en TDD resultó fundamental a la hora de agregar o cambiar requerimientos. Al igual que en un ciclo de TDD tradicional, la posibilidad de correr tests en todo momento ofrece una seguridad vital a la hora de alterar o agregar funcionalidad.

6.3 Conciliando metodologías opuestas

Las ventajas de TDD y MDWE han sido explicadas en secciones anteriores. En un principio, la idea de poder combinar ambas metodologías pareció atractiva para aprovechar todas las ventajas de las dos, es decir, la seguridad que ofrece tener tests escritos de antemano por un lado, y la facilidad y robustez que poseen los modelos por otro. No obstante, ante un primer análisis es fácil ver que conciliar TDD y MDWE es inviable, dado que si se las analiza por separado, estas dos metodologías son diametralmente opuestas. ¿Cuál es entonces el problema, y cómo se pudo resolver?

Test-Driven Development es una metodología basada exclusivamente en la escritura de código fuente. Un desarrollador, o equipo de desarrolladores que emplean TDD como práctica van a hacerlo generalmente en el marco de un proceso ágil. En este contexto, la premisa será atacar partes pequeñas del problema, de manera de ir resolviendo poco a poco las funcionalidades teniendo siempre el control total de cada línea de código que se escribe. Nada se programa sin un test, y gracias al trabajo minucioso de validar todo lo que se programa, al ir creciendo la aplicación, los riesgos de introducir errores son dramáticamente menores que en un ciclo sin tests (Maximilien & Williams, 2003).

Las metodologías dirigidas por modelos son radicalmente diferentes de TDD respecto a que se alejan lo más posible de la escritura de código. Los tests de unidad no tienen demasiado sentido en esta metodología, dado que no se puede ejercitar el SUT (System Under Test – Sistema Bajo Testeo) de una manera concreta a nivel de código, puesto que este código es derivado mecánicamente, por lo cual el resultado es imprevisible; únicamente se asegura que cumplirá con la funcionalidad modelada. Además, dada la naturaleza de las metodologías basadas en modelos, el código podría derivarse incluso en diferentes lenguajes, de modo que de querer aplicar el mismo principio de tests de unidad, habría que hacerlo sobre los modelos, pero es una práctica que no va a asegurar los mismos resultados debido a que lo que realmente se ejecuta no son los modelos en sí, sino lo que éstos derivan. Para sumar otra complicación, aún logrando hacer tests sobre modelos, tendríamos que buscar la manera hacerlos *antes* de desarrollar dichos modelos para emular la técnica de Test-First programming, fundamental en TDD. Otra característica para destacar acerca de MDWE está en que un ciclo de desarrollo tradicional de esta metodología va a abordar conjuntos de requerimientos mayores que los que normalmente se atacan por ejemplo en un *sprint* de *Scrum* (Kniberg, 2007), por mencionar una metodología ágil.

En resumen, las diferencias principales entre estas dos metodologías están claras: MDWE es una metodología de alto nivel, donde se resuelven múltiples requerimientos en cada paso y el código se deriva automáticamente, mientras que TDD está fuertemente centrada en la escritura de código, utiliza tests sobre ese código y aborda conjuntos pequeños de requisitos para ser resueltos en períodos cortos. Estas pequeñas partes de

funcionalidad se van sumando hasta obtener un prototipo o sistema completo, contrariamente a lo que sucede en MDWE, donde se comienza el diseño en un nivel más abstracto para derivar código concreto en etapas posteriores mediante transformaciones.

Llegamos a la conclusión de que estamos queriendo conciliar:

- Una metodología Top-Down con una que es Bottom-Up.
- Una metodología basada en la escritura de código fuente con otra que se basa en modelos y transformaciones.
- Una metodología que implementa en ciclos cortos vs. otra que sigue un desarrollo tradicional en cascada.

Aún así, la idea de combinar las ventajas de ambas sigue siendo atractiva. La clave estuvo en adaptar las ventajas de TDD a un ciclo de desarrollo MDWE, y no al revés. Tomando esta perspectiva, primero se pensó en cuáles son las ventajas que se querían aprovechar de TDD para poder transpolar las ideas al ciclo basado en modelos.

En primer lugar, el problema de trasladar la idea de hacer tests de unidad de antemano durante un desarrollo MDWE se resolvió cambiando el foco de los tests, cosa que resultó fundamental además para adaptar las ideas de TDD a las aplicaciones web, en contraste con las aplicaciones de escritorio. Lo que buscamos es hacer tests sobre la interacción y navegación que realiza el usuario, que son además los aspectos más característicos y fundamentales de la mayoría de las aplicaciones web. Por este motivo optamos por usar tests de interacción.

Para resolver la segunda contradicción (código fuente vs. modelos y transformaciones) simplemente elegimos usar modelos, dado que al haber resuelto el problema de los tests incluyendo tests de interacción, ya no existe un motivo por el cual sea necesario escribir código fuente.

Con respecto al tercer punto, la alternativa simple fue la de adaptar el ciclo de las metodologías dirigidas por modelos al estilo ágil. Si bien cada ciclo de MDWE tiene más sentido tomando conjuntos de requerimientos levemente mayores que los una iteración de TDD, se pudo adaptar sin problemas.

Capítulo 7
Resultados derivados y trabajo futuro

A partir de la propuesta metodológica de desarrollo de aplicaciones web aquí presentada se han generado varias inquietudes en diferentes sentidos para mejorar o completar el proceso y sus herramientas asociadas, en algunos casos se intentó contrarrestar algunas debilidades detectadas y en otros asistir el desarrollo mismo o mejorar la forma de documentar los requerimientos.

7.1 WebSpec

La metodología presentada posee dos debilidades que están relacionadas:

- La elaboración de tests de interacción puede resultar tediosa. Si bien podemos usar la técnica de captura/reproducción para no tener que escribirlos, siempre hay que programar las aserciones a mano. Más aún, si se quiere reproducir un test para extenderlo a otros casos, también hay que hacerlo manualmente.
- Los UIDs no tienen la suficiente expresividad para representar todo lo que puede suceder en una aplicación Web. Ciertas interacciones, sobre todo las que tienen que ver con funcionalidad RIA, quedan relegadas en otro tipo de documentación, mientras que sería ideal poder capturar todo en un mismo diagrama.

Una idea que surgió para atacar el problema de la escritura manual de tests de interacción fue la de derivar los tests a partir de los posibles caminos de navegación que se representaban en los UIDs. Esto tenía sentido dado que toda la interacción que podía llevarse a cabo entre los usuarios y la aplicación estaba representada en estos diagramas, por lo cual si se lograba generar un test por cada uno de estos caminos de navegación, generando además baterías de datos para simular la entrada del usuario, estaríamos cubriendo una buena parte de la funcionalidad.

El segundo problema, es decir la falta de expresividad de los diagramas UID simplemente requirió de la extensión del lenguaje para poder cubrir las nuevas formas de interacción, y dado que necesitábamos también resolver el problema de la generación automática de tests, se podría aprovechar para tener esto en mente y generar un nuevo lenguaje que cubra con todas las expectativas y que sea lo suficientemente formal como para poder derivar tests de interacción a partir de él.

Teniendo todas las razones recién explicadas en mente surgió WebSpec (Luna, Burella, Grigera, & Rossi, 2010), un diagrama de interacción basado en UIDs pero con mayor poder expresivo, y la capacidad de denotar precondiciones, postcondiciones e invariantes. WebSpec es un DSL (Domain Specific Language – Lenguaje de Dominio Específico) que permite especificar navegación, interacción y aspectos de interfaz de usuario de una manera suficientemente formal como para derivar tests de interacción a partir de sus diagramas.

Una interacción (la contraparte de una página web en la etapa de requerimientos) representa un punto donde el usuario puede interactuar con la aplicación utilizando sus componentes de interfaz. Las interacciones tienen un nombre (único por diagrama) y pueden tener componentes como: etiquetas, listas, botones, *radio buttons*, *check boxes* y paneles. Las etiquetas definen el contenido mostrado por una interacción. Las interacciones se representan gráficamente con un rectángulo redondeado que contiene el nombre de la interacción y las componentes. Todo diagrama WebSpec debe tener una interacción inicial que se denota con líneas punteadas.

La sintaxis completa para los diagramas WebSpec puede verse en detalle en la figura 5.2.

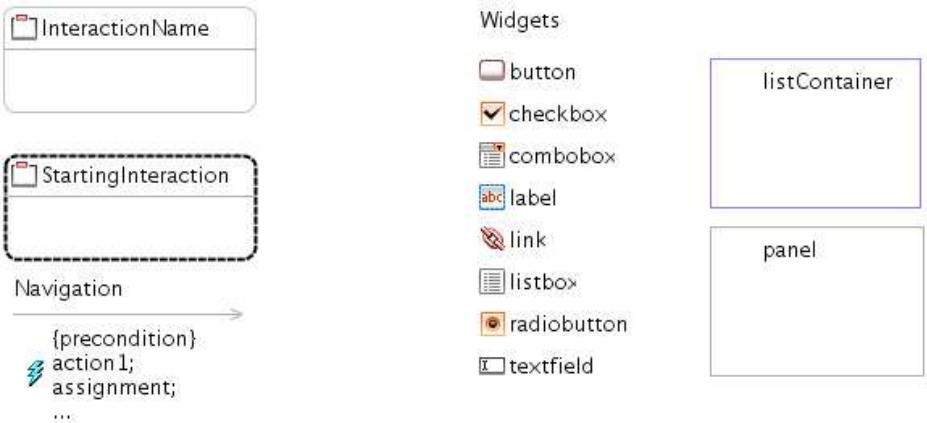


Figura 5.2: Elementos de notación de diagramas WebSpec.

Todo diagrama WebSpec se relaciona con un mockup. De esta manera es posible luego generar los tests de interacción para que puedan correrse sobre dicho mockup. A

través de identificadores HTML corrientes se puede establecer una relación entre lo que se especifica en un diagrama WebSpec y lo que se representa en el mockup asociado. Únicamente es necesario que cada elemento del mockup tenga una manera unívoca de ser referenciado.

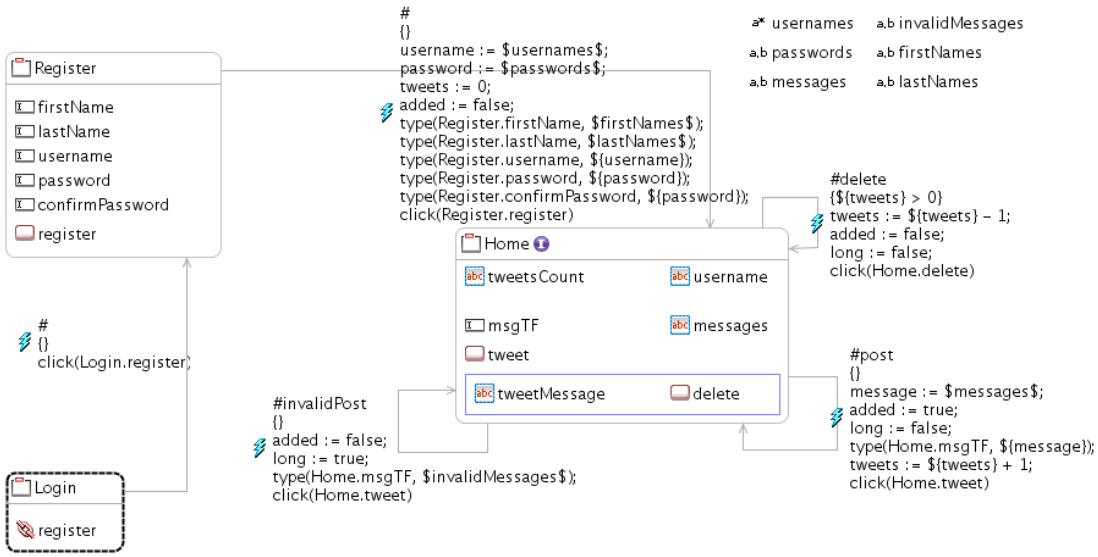


Figura 5.3: Ejemplo de diagrama WebSpec.

Los invariantes son predicados booleanos que deben cumplirse en todo momento. Cada interacción tiene un invariante que especifica qué propiedad debe ser satisfecha. La figura 5.3 muestra un ejemplo de un diagrama WebSpec para una aplicación similar a Twitter, en la cual los usuarios pueden dejar mensajes breves en el estilo de un blog. El requerimiento que aquí se representa es la publicación de un mensaje (llamado *tweet*) y tiene tres interacciones: **Login**, **Register** y **Home**. La interacción Home define un invariante (marcado con el ícono I cerca del nombre de la interacción):

```
Home.username = ${username} &&
Home.tweetsCount = ${tweets} &&
${long} -> Home.messages = "Invalid message"
```

El invariante especifica que el contenido de la etiqueta **username** debe ser igual al de la variable **${username}** y que el contenido de la etiqueta **tweetsCount** debe ser igual al de la variable que cuenta los tweets, y que si la variable **${long}** es verdadera, entonces el contenido de la etiqueta **messages** debe ser igual a "Invalid message".

Una navegación desde una interacción hacia la otra puede ser transitada si su precondición se mantiene al ejecutar una secuencia de acciones como clickear un botón, agregar texto en un campo, etc. Al igual que los invariantes, las precondiciones pueden referirse a variables previamente declaradas en el diagrama. Por ejemplo, la navegación "delete" en la figura 5.3 tiene la precondición **${tweets}>0**. Las navegaciones se representan gráficamente en el diagrama WebSpec como líneas grises mientras que su

nombre, precondiciones y acciones se muestran como etiquetas sobre ellas. Las acciones se escriben en un DSL intuitivo con la siguiente sintaxis:

var := expr | actionName(arg1,… argn)

Las navegaciones corrientes mediante vínculos se representan sin precondiciones, o con una precondición que siempre es verdadera, y con una sola acción, como se ve por ejemplo el link "**register**" en la interacción "**Login**" de la figura 5.3. En la misma figura se puede ver un ejemplo de una secuencia un poco más compleja de acciones, como es **invalidPost**.

```
1. added := false;
2. long := true;
3. type(Home.msgTF, $invalidMessages$);
4. click(Home.tweet);
```

Las primeras dos líneas (1-2) asignan valores constantes a las variables **added** y **long**. Luego, en la línea 3 se ingresa un texto utilizando el generador **invalidMessages** al campo **msgTF** y finalmente en la línea 4 se presiona el botón **tweet**.

WebSpec permite especificar propiedades generales como "un error debe mostrarse si el usuario intenta publicar un mensaje con más de 150 caracteres" usando generadores. Siguiendo la idea de QuickCheck (Claessen & Hughes, 2000), se extrae información utilizada para especificar requerimientos de interacción en generadores. Si una propiedad en un diagrama WebSpec es verdadera, entonces debe ser verdadera para todo elemento que pueda ser generado por un generador. Un generador es una función que puede ser llamada desde las acciones dentro de las navegaciones (como el visto anteriormente $invalidMessages$) y genera información. Por ejemplo en la figura 5.3 se pueden ver seis generadores: usernames, passwords, messages e invalidMessages, firstNames, lastNames. El generador invalidMessages genera strings de tamaño mayor a 150, de manera que cuando la navegación invalidPost se activa, un texto inválido intentará ser tipeado, y dado que la variable **long** será verdadera en dicho caso, se deberá mostrar un mensaje de error en la etiqueta messages.

Para los requerimientos de las aplicaciones web que tienen funcionalidad oculta, es decir aquella que no es activada o percibida directamente por un usuario, pueden agregarse notas al diagrama WebSpec o conectar navegaciones directamente con Use Cases o User Stories. Por ejemplo, si es necesario enviar un email de notificación automático cuando un usuario publica un mensaje, se puede agregar fácilmente una nota sobre la navegación "post".

7.2 Usabilidad

Una de las ideas principales detrás de la metodología propuesta es la de ir cumpliendo con todos lo requerimientos en cada iteración a medida que se avanza con el desarrollo, y validar estos requerimientos con los clientes.

Por otra parte, la usabilidad es una característica fundamental que debe estar presente en toda aplicación web, sobre todo teniendo en cuenta que en la mayoría de los casos y a diferencia de las aplicaciones de escritorio, un usuario frustrado ante una aplicación web simplemente cerrará el navegador o elegirá otra opción rápidamente, de serle posible.

Aprovechando entonces las características de la metodología y la necesidad de generar aplicaciones usables en todo momento, consideramos que sería una buena práctica involucrar los requerimientos relacionados a la usabilidad en las primeras etapas del desarrollo, de la misma manera que lo hacemos con los requisitos de presentación e interacción al usar mockups y tests de interacción antes de comenzar con la implementación.

En un trabajo con otros autores (Luna, Grigera, Rossi, Panach, & Pastor, 2009) se profundizó en la idea de trabajar con los requisitos de usabilidad en la metodología aquí propuesta. Básicamente, consideramos que los requisitos de usabilidad son requisitos semejantes a los convencionales, pero que capturan las características necesarias para construir un sistema usable ya sean que tengan que ver con la funcionalidad pura de la aplicación o no.

Al considerar los requisitos de usabilidad como requisitos convencionales, también generamos tests para comprobar que tales requisitos se cumplen, y de la misma manera que sucede con los requisitos convencionales, los tests pueden seguir corriéndose en las iteraciones subsiguientes de desarrollo para comprobar que la usabilidad no ha sido comprometida al incorporar nuevas funcionalidades a la aplicación web.

Entre todos los posibles requisitos de usabilidad, resultan de particular interés aquellos que tienen implicancias en la arquitectura del sistema. Estos requisitos son descriptos en la literatura como Características Funcionales de Usabilidad (originalmente FUFs – Functional Usability Features) (Juristo, Moreno, & Sanchez-, 2007). Algunos ejemplos de estos requisitos son:

- Proveer la posibilidad de **cancelar** una operación: puede requerir que la navegación deba modificarse, o incluso el modelo de dominio.
- Proveer la posibilidad de **deshacer** una operación: al igual que con la posibilidad de cancelar, puede modificar el modelo de dominio, y modifica definitivamente la navegación.
- Proveer información de estado (**feedback**): en este caso la navegación también se puede ver afectada. Por ejemplo podríamos querer informar el estado de la carga de un archivo o de una operación que toma normalmente mucho tiempo.

A su vez, cada FUF tiene un conjunto de subtipos llamados Mecanismos de Usabilidad (*Usability Mechanisms*). Cada Mecanismo de Usabilidad es en realidad una

variante del FUF al que está ligado. Por ejemplo, para el FUF **feedback** los Mecanismos de Usabilidad pueden ser:

1. Información de **estado** del sistema: informa diferentes parámetros del estado interno de la aplicación.
2. Información de **interacción**, que reporta al usuario que una acción determinada solicitada por él se encuentra en proceso.

Para no depender exclusivamente de la mano de un experto en usabilidad al momento de detectar los requerimientos de usabilidad, se utilizaron las guías (o catálogos de buenas prácticas) del trabajo que explica los FUFs mencionado anteriormente. Estas guías ayudan a los desarrolladores a comprender las implicancias de los requerimientos de usabilidad en la arquitectura del sistema y a conocer la manera de capturar y especificar requerimientos de usabilidad para un sistema.

7.3 Accesibilidad

En un trabajo similar al explicado en la sección anterior relacionado con la usabilidad, se exploró junto a otros autores la posibilidad de hacer que la metodología propuesta favorezca la generación de aplicaciones web *usables para la accesibilidad* (Medina Medina, Rossi, Garrido, & Grigera, 2010).

La idea de una aplicación usable para la accesibilidad está orientada a los usuarios con diferentes discapacidades, particularmente discapacidades visuales. Este tipo de usuarios suele recurrir a herramientas que los asisten para navegar la web, como lectores de pantalla (que literalmente leen los contenidos web con una voz sintetizada) o navegadores especiales que pre-procesan el contenido web para simplificar la presentación. Para que estas herramientas funcionen correctamente, los sitios deben ser idealmente accesibles conforme a las reglas WCAG del consorcio W3C[14]. No obstante, aún cumpliendo rigurosamente con todas las reglas en el máximo nivel (AAA) la aplicación puede no estar lista para el uso óptimo por parte de usuarios discapacitados.

Para poder mejorar y facilitar el acceso universal, se propuso aplicar transformaciones a las aplicaciones web para que provean mejoras tanto en navegación como en presentación, generalmente simplificando dichos aspectos. Por supuesto, en algunos casos es posible que estas simplificaciones perjudiquen a los usuarios regulares, por lo tanto lo ideal es mantener dos aplicaciones en paralelo para que el acceso y la usabilidad sean realmente universales.

Si bien la idea general se parece al trabajo presentado en la sección 7.2, el enfoque metodológico es diferente. En este caso la propuesta no consiste únicamente en desarrollar teniendo los requisitos de usabilidad funcionales en cuenta (a los que agregamos requisitos usabilidad no funcionales y otros de accesibilidad), sino en aplicar transformaciones a la navegación y presentación apuntando a mejorar la usabilidad para la accesibilidad llamados refactorings, dado que tienen un objetivo similar a los que propone Fowler (Fowler, Beck, Brant, Opdyke, & Roberts, 1999). En el caso de los últimos,

[14] http://www.w3.org/TR/WCAG20/

la idea es mejorar la modularidad y legibilidad del código preservando la funcionalidad, en el caso de los (web) refactorings que presentamos en nuestro trabajo, la idea también es preservar la funcionalidad pero introduciendo mejoras en la navegación y presentación. Una muestra del catálogo de refactorings puede verse en las tablas 7.1 y 7.2, donde se muestran algunos refactorings de navegación y presentación respectivamente.

Refactoring	Propósito
Dividir nodo	Limpiar un nodo complejo, creando en su lugar varios sub nodos, cada uno con un subconjunto de contenido cohesivo, haciendo que sean más legibles que el nodo original.
Unir nodos	Combinar dos nodos altamente relacionados. Sólo puede aplicarse cuando haya una clara correspondencia en los contenidos de ambos nodos, y no se genere un nodo complejo.
Agregar vínculo	Crear un nuevo vínculo entre dos nodos relacionados, facilitando el acceso mutuo entre ellos y eliminando la necesidad de transitar el nodo padre para pasar de uno al otro.

Tabla 7.1: Refactorings de navegación.

Refactoring	Propósito
Distribuir menú	Hacer que un menú con acciones que se aplican a una selección de elementos desaparezca, dejando en su lugar vínculos asociados a cada elemento para poder aplicar la acción localmente.
Reemplazar gráficos por texto	Reemplazar imágenes y gráficos indicativos por el texto equivalente para facilitar la lectura.
Eliminar imágenes estéticas	Eliminar contenido visual que no agrega información relevante, cumpliendo un rol estrictamente estético en la página.

Tabla 7.2: Refactorings de presentación.

7.4 Manejo de cambios

Una de las motivaciones principales de este trabajo fue la de agilizar el proceso de desarrollo de aplicaciones web en el momento de la **evolución** o **cambios en los requerimientos**. Dado que el proceso es inherentemente iterativo, y que en cada iteración se resuelve un conjunto acotado de requerimientos, resultó atractiva la idea de poder llevar una traza de dichos requerimientos y las consecuencias que implicaba su implementación en todos los niveles de la aplicación: presentación, navegación y modelo de negocio.

Habiendo avanzado ya la noción de diagramas WebSpec, e incluso existiendo una herramienta para modelarlo, la parte de capturar y documentar los requerimientos con artefactos de software está cubierta. Para ir un poco más allá y capturar toda la iteración, se introdujo en un trabajo junto con otros autores la idea de **change objects** (Burella, Rossi, Robles Luna, & Grigera, 2010), tomada parcialmente del trabajo de ChangeBoxes de Oscar Niestrasz (Denker et al., 2007).

Los change objects son objetos que encapsulan el cambio realizado en la aplicación. A través de estos objetos podemos:

- Mantener una **traza** de los cambios aplicados en cada iteración, y tener la posibilidad de saber, para cada cambio, desde el diagrama o diagramas WebSpec que fueron agregados o modificados hasta el código o cambio en los modelos que se introdujo en la implementación.
- Proveer la posibilidad de **podar** la cantidad de tests que se corren luego de cada iteración. En la metodología aquí propuesta, al igual que en TDD tradicional, la idea es siempre correr *todos* los tests para asegurarnos de que no hay funcionalidad previa comprometida al agregar nuevos cambios. Esto puede volverse tedioso a medida que el número de tests crece. Por esto, y gracias a los change objects, podemos llevar cuenta de cuáles son los tests que tiene sentido correr al introducir un cambio, de esta manera reduciendo la cantidad necesaria, al menos durante el desarrollo.
- Podemos **automatizar** algunos cambios. Dado que los change objects son ciudadanos de primera clase en la implementación, podemos hacer que los cambios se apliquen automáticamente a partir de las modificaciones a los diagramas WebSpec. Esta noción se lleva muy bien con la idea de refactorings explicada en la sección anterior.

7.5 Trabajo futuro

Existen muchas direcciones sobre las cuales puede evolucionar la metodología de desarrollo presentada en este trabajo.

En primer lugar, para comprobar de una manera más contundente las ventajas de seguir este proceso, se está desarrollando un nuevo experimento similar al presentado en el capítulo 4, pero con un mayor número de individuos y un mejor trabajo de preparación para evitar que los resultados muestren cualquier tipo de sesgo. Además se planea tener

un mejor control sobre el desarrollo de cada individuo, para obtener mejor información respecto del tiempo insumido y las dificultades surgidas durante la implementación.

Otro sentido en el que se está planificando avanzar es en la creación de herramientas para soportar los diferentes artefactos de la metodología. Por ejemplo, WebSpec ya tiene su propia herramienta, como plugin del ambiente de desarrollo Eclipse. También hay un desarrollo hecho en Seaside[15] (un framework para desarrollar aplicaciones web basado en el lenguaje Smalltalk) para soportar la administración de change objects y la automatización de algunos de ellos para el desarrollo en dicho framework. Se está trabajando en extender este manejo a otros lenguajes de programación y herramientas MDWE.

Siguiendo la línea de los resultados derivados, también se quiere continuar con el trabajo acerca de generar versiones accesibles y usables de una aplicación web automáticamente. Si bien las ideas ya han sido presentadas, hay trabajo por hacer respecto de cómo mantener diferentes versiones universalmente accesibles de la misma aplicación con costo mínimo.

[15] http://seaside.st

Bibliografía

- Alles, M., Crosby, D., Harleton, B., Pattison, G., & Erickson, C. (2006). Presenter first: organizing complex GUI applications for test-driven development. *Agile Conference, 2006*, 276-288.

- Beck, K. (2002). *Test Driven Development: By Example* (p. 240). Addison-Wesley Professional.

- Burella, J., Rossi, G., Robles Luna, E., & Grigera, J. (2010). *Agile Processes in Software Engineering and Extreme Programming*. (A. Sillitti, A. Martin, X. Wang, & E. Whitworth)*Agile Processes in Software Engineering and Extreme Programming*, Lecture Notes in Business Information Processing (Vol. 48, pp. 220-225). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-13054-0.

- Cachero, C., & Koch, N. (n.d.). Navigation Analysis and Navigation Design in OO-H and UWE. *Science And Technology*, (April 2002).

- Ceri, S., Fraternali, P., & Bongio, A. (2000). Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, *33*(1-6), 137-157. doi: 10.1016/S1389-1286(00)00040-2.

- Claessen, K., & Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, *35*(9), 268-279. doi: 10.1145/357766.351266.

- Denker, M., Gîrba, T., Lienhard, A., Nierstrasz, O., Renggli, L., Zumkehr, P., et al. (2007). Encapsulating and exploiting change with changeboxes. *ACM International Conference Proceeding Series; Vol. 286*, 25-49.

- Eleftherakis, G., & Cowling, A. (2003). An agile formal development methodology. *of the 1st South-East European Workshop on Formal*, (November), 36-47.

- Evans, E. (2003). *Domain-driven Design: Tackling Complexity in the Heart of Software* (p. 560). Addison Wesley.

- Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code (Object Technology Series)* (p. 464). Addison Wesley.

- Jacobson, I. (1992). *Object Oriented Software Engineering: A Use Case Driven Approach* (p. 552). Addison-Wesley Professional.

- Janzen, D., & Saiedian, H. (2008). Does Test-Driven Development Really Improve Software Design Quality? *IEEE Software*, *25*(2), 77-84. doi: 10.1109/MS.2008.34.

- Jeffries, R., Anderson, A., & Hendrickson, C. (2000). *Extreme Programming Installed* (p. 288). Addison-Wesley Professional.

- Juristo, N., Moreno, A., & Sanchez-, M. (2007). Guidelines for eliciting usability functionalities. *IEEE Transactions on*, *33*(11), 744-758. doi: 10.1109/TSE.2007.70741.

- Kniberg, H. (2007). *Scrum and XP from the Trenches (Enterprise Software Development)* (p. 140). Lulu.com.

- Koch, N., Knapp, A., Zhang, G., & Baumeister, H. (2005). Chapter 7 UML-BASED WEB ENGINEERING An Approach Based on Standards. *Engineering*.

- Larman, C. (2003). *Agile and Iterative Development: A Manager's Guide* (p. 368). Addison-Wesley Professional.

- Luna, E. R., Burella, J., Grigera, J., & Rossi, G. (2010). A flexible tool suite for change-aware test-driven development of web applications. *International Conference on Software Engineering*, 297-298.

- Luna, E., Grigera, J., Rossi, G., Panach, J., & Pastor, O. (2009). Introducing Usability Requirements in a Test/Model-Driven Web Engineering Method1. *ceur-ws.org*, 28-39.

- Mackinnon, T., Freeman, S., & Craig, P. (2001). Endo-testing: unit testing with mock objects. *Extreme programming examined*.

- Maximilien, E., & Williams, L. (2003). Assessing test-driven development at IBM. *of the 25th International Conference on*, 564-569. Ieee. doi: 10.1109/ICSE.2003.1201238.

- McDonald, A., & Welland, R. (2003). Agile web engineering (AWE) process: multidisciplinary stakeholders and team communication. *Web Engineering*, 515-518.

- Medina Medina, N., Rossi, G., Garrido, A., & Grigera, J. (2010). Refactoring for Accessibility in Web Applications. In *Interacción 2010*. Valencia.

- Pastor, O., Abrahão, S., & Fons, J. (2001). An Object-Oriented Approach to Automate Web Applications Development. *Information Systems*, 16-28.

- Pastor, Ó. (2006). From Extreme Programming to Extreme Non-programming: Is It the Right Time for Model Transformation Technologies? In *Database and Expert Systems Applications* (p. 64–72). Springer.

- Pipka, J. (2009). Test-driven web application development in java. *Components, Architectures, Services, and Applications*.

- Rasmusson, J. (2003). feature Introducing XP into Greenfield Projects :. *Ieee Software*, 21-28.

- Robles Luna, E., Grigera, J., & Rossi, G. (2009). *Bridging Test and Model-Driven Approaches in Web Engineering*. (M. Gaedke, M. Grossniklaus, & O. Díaz)*Web Engineering*, Lecture Notes in Computer Science (Vol. 5648, pp. 136-150). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-02818-2.

- Rossi, G., & Schwabe, D. (2008). *Web Engineering: Modelling and Implementing Web Applications*. (G. Rossi, O. Pastor, D. Schwabe, & L. Olsina)*Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series (pp. 109-155). London: Springer London. doi: 10.1007/978-1-84628-923-1.

- Snyder, C. (2003). *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces (Interactive Technologies)* (p. 408). Morgan Kaufmann.

- Vilain, P., Schwabe, D., & Souza, C. S. (n.d.). A Diagrammatic Tool for Representing User Interaction in UML.