

Esteban Robles Luna

Agile Managing of Web requirements with WebSpec

– PhD. Thesis –

Advisors: Gustavo Rossi, Irene Garrigos Fernandez

Depto. Lenguajes y Sistemas Informáticos

Universidad de Alicante

Facultad de Informática

Universidad Nacional de La Plata

*No creas lo que tus ojos te dicen. Sólo muestran limitaciones. Mira con tu entendimiento,
descubre lo que ya sabes, y hallarás la manera de volar.*

Richard Bach. Juan Salvador Gaviota.

*Don't believe what your eyes are telling you. All they show is limitation. Look with your
understanding, find out what you already know, and you'll see the way to fly.*

Richard Bach. Jonathan Livingston Seagull.

Agradecimientos

Cuando decidí aventurarme en el doctorado no estaba muy seguro de cómo iba a terminar. Aunque los que me conocen dirán que (en gral) lo que comienzo lo termino, la verdad es que no siempre es así. En esta aventura me acompañaron y se sumaron muchas personas que me dieron apoyo, contención y por sobre todo tiempo y paciencia. A todos aquellos que lean estas líneas les quiero decir GRACIAS sin olvidarme de ninguno de udes. Sin embargo, algunos de udes merecen un reconocimiento más afectuoso.

A mis viejos y mi hermano por todo el apoyo que me dieron y porque me formaron. Gracias por la paciencia que me tienen cuando digo alguna locura de irme a vivir a otro lugar o ir a un congreso por un tiempo determinado.

A Sole, que me ha acompañado, tolerado y amado durante todo este tiempo. Me alegro que podamos compartir juntos estas aventuras locas que se me están ocurriendo estos últimos meses; espero que las experiencias y mi compañía lo valgan.

A mis directores Gustavo e Irene por toda la paciencia, ayuda y consejo en mi trabajo y mi vida. Me siento afortunado de haber trabajado con Udes.

A mi familia más grande (tíos y primos) en particular a mis tíos Carlos y Alfredo y a mi tía Marisa con los cuales he tenido una relación más estrecha y les tengo un gran afecto.

A mis amigos, en particular a Pablo, Cristian, Santi y Hernán por todos los momentos que hemos vivido y porque udes. me han ayudado a empujar y alcanzar lo que hoy estoy logrando.

A varios de mis profes del secundario: Julio Paladino, Roxana Rodríguez, Silvia García y Liliana Boemo; ellos marcaron mi vida sustancialmente desde un punto de vista profesional a muy temprana edad:

1. A Julio: vos me enseñaste que las cosas se pueden lograr, que teniendo un plan sensato y haciendo algunos sacrificios las cosas salen. Aunque no lo crean, no estoy hablando de ninguna materia/asignatura intelectual, sino del atletismo.
2. A Roxana y a Liliana: Porque fueron mis guías matemáticas; las que me acompañaron en toda competencia donde quería ir y quedaba clasificado, las que me formaron en muchas áreas donde el secundario se quedaba corto y yo quería alcanzar.
3. A Silvia: Porque en el momento donde me sentí más débil y no creía en mí, ella estuvo ahí, creyó en mí y me hizo revalorizarme. Esto fue bien al comienzo del secundario y aunque fue un hecho puntual, todavía lo recuerdo.

A mis compañeros del LIFIA y el DLSI por todos los momentos que hemos compartido. En particular a Andres F, Julian G, Juan B, Norberto M, Paul H, Octavio G, Jose Alfonso A, Nahuel L, Emiliano P, Santi, Matias R, Matias U, Seba P.E (espero no olvidarme de ninguno). Con muchos de udes compartimos almuerzos, alguna mateada en el laboratorio, papers, chats o llamadas de skype. Gracias chicos!

A mis compañeros y amigos de Globant y Mulesoft que soportaron las “locuras” de hablar de papers y de discutir ideas más conceptuales. He pasado momentos muy agradables con udes y he aprendido cosas de la vida que son más difíciles de encontrar en ámbitos no laborales.

Acknowledgments

When I decided to start the adventure of a PhD I was not totally sure about its ending. Though people who know me say that (in general) everything I start, I finish, the truth is that it is not always like that. In this adventure I was not alone and several people joint me to give their support and above all, time and patient. I want to say a big THANK YOU to all of them.

To my parents and my brother: for all their support and because they helped me to be a better person. Thanks for your patient and comprehension when I say something insane like living in a different country.

To Sole: because of her support, tolerance and love during this time. I'm glad that we can share this crazy adventures that we are living in the last couple of months; I hope that the experience and my presence worth it.

To my directors: Gustavo and Irene, because of their patience, help and advice in work and life. I feel lucky of working with you.

To my big family (aunts, uncles and cousins), in particular to my uncles Carlos and Alfredo and to my aunt Marisa with whom I have a close affective relationship.

To my friends, in particular to Pablo, Cristian, Santi and Hernan for all the moments we have lived together and because you helped me to push and reach this thesis.

To several of my high school professors: Julio Paladino, Roxana Rodríguez, Silvia García y Liliana Boemo; because they are important persons in my life and their presence affected my professional thinking:

1. To Julio: because you taught me that objectives could be reached with a reasonable plan and doing small sacrifices. Although you may not believe it, I'm not talking about any intellectual subject, but of Athletics.
2. To Roxana and Liliana: Because they were my mathematics guiders which support me in any mathematics competition and the ones who taught me in several areas where the contents of high school were not enough.
3. To Silvia: Because in the moment where I felt weak and I did not believe in myself, she was there to help me reevaluate. Though this happened at the beginning of high school, it was a fact I still remember.

To my partners of LIFIA and DLSI: for all the moments that we have lived together. In particular to: Andres F, Julian G, Juan B, Norberto M, Paul H, Octavio G, Jose Alfonso A, Nahuel L, Emiliano P, Santi, Matias R, Matias U, Seba P.E (I hope not to missed any of you). We have shared lunch, mate at the lab, papers, chats and skype calls. Thanks Guys!

To my partners and friends of Globant and Mulesoft: because of their support and talks about papers and conceptual ideas. We have lived pleasant moments and I have learnt things about life that are more difficult to find outside work.

Preface

Web application development is a complex and time consuming process that involves different stakeholders (ranging from customers to developers); these applications have some unique characteristics like navigational access to information, sophisticated interaction features, etc. However, there have been few proposals to represent those requirements that are specific to Web applications. Consequently, validation of requirements (e.g. in acceptance tests) is usually informal, and as a result troublesome.

To overcome these problems, this PhD Thesis proposes WebSpec, a domain specific language for specifying the most relevant and characteristic requirements of Web applications: those involving interaction and navigation. We describe WebSpec diagrams, discussing their abstraction and expressive power.

As part of this work, we have created a test driven model based approach called WebTDD that gives a good framework for the language. Using the language with this approach we have test several of its features such as automatic test generation, management of changes in requirements, and improving the understanding of the diagrams through application simulation.

This PhD Thesis is composed of a set of published and submitted papers. In order to write this PhD Thesis as a collection of papers, several requirements must be taken into account as stated by the University of Alicante. With regard to the content of the PhD Thesis, it must specifically include a summary which is devoted to the description of initial hypotheses, research objectives, and the collection of publications itself, thus justifying its coherence. It should be underlined that this summary of the PhD Thesis must also include research results and final conclusions. This summary corresponds to part I of this PhD Thesis (chapter 1 has been written in Spanish while chapter 2 is in English).

This work has been partially supported by the following projects: MANTRA (GV/2011/035) from Valencia Ministry, MANTRA (GRE09-17) from the University of Alicante and by the MESOLAP (TIN2010-14860) project from the Spanish Ministry of Education and Science.

Contents

Part I Summary

1	Síntesis en Castellano	3
1.1	Tesis Doctoral como Compendio de Artículos	3
1.1.1	Publicaciones Pertenecientes a esta Tesis Doctoral	3
1.1.2	Otras Publicaciones en Congresos Internacionales	5
1.2	Objetivos de Investigación e Hipótesis Inicial	6
1.3	Resumen del Contenido de la Tesis Doctoral	7
1.3.1	WebTDD	7
1.3.2	WebSpec	8
1.3.3	Implementación	15
1.4	Conclusiones	20
	References	20
2	Summary in English	23
2.1	PhD Thesis as a Collection of Papers	23
2.1.1	Publications Included in this PhD Thesis	23
2.1.2	Other publications in International conferences	25
2.2	Research Objectives and Initial Hypotheses	26
2.3	PhD Thesis in a Nutshell	27
2.3.1	WebTDD	27
2.3.2	WebSpec	28
2.3.3	Implementation	34
2.4	Conclusions	39
	References	40

Part II PhD Thesis as a Collection of Papers

3	A context for WebSpec: The WebTDD approach	43
4	Capture and Evolution of Web requirements using WebSpec	87
5	Integrating an early phase of requirements to WebSpec	105
6	Specifying personalizable and accessible web applications with WebSpec .	133
7	Change management and tool support for WebSpec	159

Part III Appendix: Papers Already Submitted

XIV Contents

A	WebSpec: a Visual Language for Specifying Interaction and Navigation Requirements in Web Applications	173
----------	--	------------

Part I

Summary

Síntesis en Castellano

La presente tesis doctoral se ha realizado mediante la modalidad de compendio de artículos. Por tanto, este capítulo está dedicado a describir los objetivos, hipótesis y el conjunto de trabajos que forman parte de la tesis, quedando justificada su unidad temática. Cabe destacar que en este capítulo inicial también se sintetiza el contenido científico de la tesis, presentando un resumen global de los resultados obtenidos así como de las conclusiones finales. Por último, debo resaltar que el contenido de este capítulo ha sido escrito en castellano, mientras que el capítulo siguiente corresponde a su traducción en inglés.

1.1 Tesis Doctoral como Compendio de Artículos

Los requisitos que debe cumplir una tesis doctoral para ser realizada en la Universidad de Alicante mediante un compendio de publicaciones fueron definidos por el Pleno de la Comisión de Doctorado de fecha 2 de marzo de 2005. A continuación, se exponen aquellos directamente relacionados con el contenido de la tesis:

1. *“La tesis debe incluir una síntesis, en una de las dos lenguas oficiales de esta Comunidad Autónoma, en la que se presenten los objetivos, hipótesis, los trabajos presentados y se justifique la unidad temática.”*
2. *“Esta síntesis debe incorporar un resumen global de los resultados obtenidos, de la discusión de estos resultados y de las conclusiones finales. Esta síntesis deberá dar una idea precisa del contenido de la tesis.”*
3. *“Los trabajos deben ser publicados, o aceptados para la publicación, con posterioridad al inicio de los estudios de doctorado. Los artículos en periodo de revisión pueden formar parte de la tesis como apéndices del documento, que debe presentarse adjunta a los artículos publicados.”*

Por consiguiente, con el fin de cumplir estos requisitos, la estructura de esta tesis queda definida en tres partes bien diferenciadas. La parte I consiste en un capítulo de síntesis en castellano (capítulo 1) y su correspondiente versión en inglés (capítulo 2). La parte II presenta el conjunto de artículos publicados que forman el contenido principal de esta tesis doctoral. La parte III presenta un artículo en proceso de revisión.

1.1.1 Publicaciones Pertenecientes a esta Tesis Doctoral

Se ha seleccionado un conjunto de los artículos de investigación publicados para que formen parte de esta tesis doctoral por dos razones: (i) su contribución científica y (ii) la relevancia de dichas publicaciones. Estas publicaciones se describen brevemente en esta sección.

Capítulo 3

Robles Luna E., Grigera J., Rossi G. *Bridging Test and Model Driven Approaches in Web Engineering. Proceedings of 9th International Conference on Web Engineering (ICWE 2009).* 2009. San Sebastian, Spain. Acceptance rate: 24%. Core C.

Robles Luna E., Panach J.I., Grigera J., Rossi G., Pastor O. *Incorporating Usability Requirements in a Test/Model-Driven Web Engineering Approach. Journal of Web Engineering (JWE).* 2010. Impact factor: 0.531. JCR.

Este trabajo describe una metodología de desarrollo de aplicaciones Web en la cual los tests juegan un papel fundamental. Estos dirigen el desarrollo indicando que parte de la funcionalidad requerida no ha sido implementada (al igual que en las metodologías dirigidas por tests [10]). Sin embargo, a diferencia de las metodologías dirigidas por tests en las cuales el principal objeto de desarrollo es el código; en esta se utiliza un desarrollo basado en modelos en la cual los modelos abstraen pero **no** dirigen el desarrollo.

La metodología presentada en este capítulo define un buen marco de trabajo para que WebSpec (el principal aporte de esta tesis) sea utilizado. En particular porque en estas metodologías no existe una traducción automática de requisitos a tests.

Capítulo 4

Robles Luna E., Garrigos I., Grigera J., Winckler M. *Capture and Evolution of Web requirements using WebSpec. Proceedings of 10th International Conference on Web Engineering (ICWE 2010).* Vienna, Austria. Acceptance rate: 20%. Core C.

En este capítulo se presenta el lenguaje de dominio específico que es la parte central de esta tesis. Se muestra su definición y su uso en las diferentes actividades de un ciclo de desarrollo. Aunque, Webspec fue inicialmente pensado para ser utilizado con el enfoque presentado en el capítulo anterior, en este capítulo también se muestra como puede ser utilizado en una metodología unificada.

Capítulo 5

Robles Luna E., Garrigos I., Mazon J-N., Trujillo J., Rossi G. *An i*-based Approach for Modeling and Tesing Web Requirements. Journal of Web Engineering (JWE).* 2010. Impact factor: 0.531. JCR.

Algunas metodologías de desarrollo utilizan una etapa temprana en donde se definen los objetivos y tareas del sistema/organización. Muchas veces se utiliza algún lenguaje para describir estas relaciones como es el caso de i*. En este capítulo se muestra como se puede utilizar WebSpec junto con i* para modelar los requisitos web. Al utilizarlos conjuntamente, se pueden validar en forma semi automática que los objetivos descritos en el modelo de i* sean implementados correctamente en la aplicación.

Capítulo 6

Medina, N. M., Burella, J., Rossi G., Grigera J., **Robles Luna E.** *An Incremental Approach for Building Accessible and Usable Web Applications. Proceedings of the 11th International Conference on Web Information System Engineering (WISE 2010).* Hong Kong, China. Acceptance rate: 18.8%. Core A.

Robles Luna E., Garrigos I., Rossi G. *Capturing and Validating Personalization Requirements in Web Applications. Proceedings of the 1st Workshop on The Web and Requirements*

Engineering (WeRE 2010). Sydney, Australia.

En este capítulo se presenta la utilización de WebSpec para especificar requisitos no funcionales como son la accesibilidad y la personalización de las aplicaciones Web. En cada caso se proveen pequeñas extensiones al lenguaje base con el fin de permitir la especificación de estos requisitos en el contexto de la metodología WebTDD.

Capítulo 7

Burella J., Rossi G., Robles Luna E., Grigera J. Dealing with Navigation and Interaction Requirement Changes in a TDD-Based Web Engineering Approach. Proceedings of the 11th International Conference on Agile Software Development (XP 2010), Springer Verlag, LNCS, 2010. Trondheim, Norway. Core B.

Robles Luna E., Burella J., Grigera J., Rossi G. A Flexible Tool Suite for Change-Aware Test-Driven Development of Web Applications. Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE 2010). 2010. Cape Town, South Africa. Core A.

En este capítulo se presenta el control de cambios de WebSpec el cual permite determinar los artefactos de código afectados por un cambio. Para ello se establece una asociación entre los cambios que se establecen en los requisitos con aquellos en la implementación. Además, se presenta una demostración de la herramienta que da soporte a cada una de las características del lenguaje.

Apéndice A

Robles Luna E., Rossi G., Garrigos I. WebSpec: a Visual Language for Specifying Interaction and Navigation Requirements in Web Applications. Requirements Engineering Journal. In press. Impact factor: 0.931. JCR.

En este capítulo se presenta la evolución del lenguaje base presentado en el Capítulo 4 en donde se detalla la especificación de requisitos para aplicaciones ricas en la Web. Además se presentan los detalles referidos a la gramática del lenguaje y una extensión al caso de estudio.

1.1.2 Otras Publicaciones en Congresos Internacionales

Durante el desarrollo de esta tesis doctoral, se han publicado otros artículos que no han sido explícitamente incluidos en este documento. Sin embargo, estos trabajos forman también parte de la investigación llevada a cabo durante los estudios de doctorado y completan el trabajo de tesis doctoral:

- **Robles Luna E.**, Escalona M.J, Rossi G. A requirements metamodel for Rich Internet applications. Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFTE 2010). Athens, Greece. Acceptance rate: 9%. Core B.
- Rivero J.M., Rossi G., Grigera J., Burella J., **Robles Luna E.**, Gordillo S. From mockups to user interface models: An extensible model driven approach. Proceedings of the 6th Model-Driven Web Engineering Workshop. (MDWE 2010). Vienna, Austria.
- **Robles Luna E.**, Rossi G., Burella J., Grigera J. Incremental Usability Improvement in an Agile Approach for Web Applications. Proceedings of the 1st workshop Dealing with Usability in an Agile Domain, XP'2010 workshop. (Usability&Agile 2010), 2010. Trondheim, Norway.
- **Robles Luna E.**, Grigera J., Rossi G., Panach J. I. and Pastor O. Introducing Usability Requirements in a Test/Model-Driven Web Engineering Method. Proceedings of 8th International Workshop on Web-Oriented Software Technologies (IWOOST 2009). 2009. San Sebastian, Spain.

1.2 Objetivos de Investigación e Hipótesis Inicial

El desarrollo de aplicaciones Web es un proceso complejo y que consume mucho tiempo. A su vez involucra a equipos de desarrollo multidisciplinarios (incluyendo clientes, diseñadores gráficos, desarrolladores, aseguradores de calidad, etc.) y por lo tanto el entendimiento de la aplicación varía entre los diferentes miembros del equipo. Además, estas aplicaciones poseen algunas características únicas como acceso a la información a través de la navegación e interacciones sofisticadas lo cual hace que su desarrollo sea diferente respecto a las tradicionales aplicaciones de escritorio. Como consecuencia, podemos encontrar en la literatura dos grandes grupos para el desarrollo de aplicaciones Web: la ingeniería Web dirigida por modelos (MDWE) y las metodologías ágiles.

Por un lado, varias metodologías MDWE han sido propuestas durante los últimos 20 años [11, 16, 18, 24, 27]. Todas ellas comparten un estilo arriba-abajo [28], construyendo la aplicación Web describiendo un conjunto de modelos en diferentes niveles de abstracción:

- Modelo de Contenido (o Aplicación): define los objetos de dominio y sus relaciones.
- Modelo de Hipertexto (o Navegación): define los nodos de navegación y los enlaces que publican información especificada en los objetos del modelo de Contenido.
- Modelo de Presentación: Refina el modelo de hipertexto con una interfaz de presentación concreta con páginas y elementos de interfaz.

El proceso utilizado en estas metodologías es en general arriba-abajo entregando una aplicación Web final y utilizando transformaciones de modelo a una tecnología destino.

Por otro lado, las metodologías ágiles promueven la interacción temprana y constante con los clientes. De esta forma se comprueba continuamente que el software construido satisface sus requisitos los cuales son desarrollados en períodos de tiempo cortos. Las metodologías ágiles argumentan que las especificaciones de software deben emerger naturalmente, mejorando los prototipos existentes a lo largo del desarrollo hasta que la aplicación final es obtenida.

En resumen, mientras que las metodologías MDWE facilitan el software portable, el nivel de abstracción y la productividad, fallan en proveer interacción ágil con los clientes porque los resultados concretos son obtenidos demasiado tarde. Por otro lado, mientras que esta última característica es lograda con claridad por las metodologías ágiles, están basadas en la implementación directa y por lo tanto fallan en proveer portabilidad, abstracción y productividad mediante la generación de código automático.

De acuerdo a diversos estudios [22, 19] en la industria, la fase de captura de requisitos es una de las más importantes de cualquier metodología de desarrollo Web. Desafortunadamente, en el contexto de MDWE, los requisitos son generalmente capturados con casos de uso [17] o una modificación de ellos mientras que en las metodologías ágiles existe una tendencia a reemplazar los casos de uso con historias de usuario [20]. Respecto al poder expresivo de ambos artefactos, estos son muy **pobres para expresar** las particularidades de la Web (por ejemplo, su naturaleza de navegación e interacción). Además, la rápida evolución de las aplicaciones Web (en pocas semanas) impone restricciones adicionales para el testing continuo y en tiempo respecto a la especificación de requisitos [19] principalmente para validar que los nuevos requisitos han sido implementados correctamente sin “romper” los existentes. En este contexto, la captura y el modelado de requisitos debe ser lo suficientemente eficiente para cumplir con las restricciones de tiempo. Por lo tanto es importante que los requisitos sean **fácilmente entendidos** para proveer una evolución eficiente de la aplicación.

Tomando en cuenta estos puntos, **la hipótesis de esta tesis doctoral** es la mejora del desarrollo de aplicaciones Web mediante:

- Una especificación formal de requisitos que automatice su validación, semi automatice la derivación de la aplicación y ayude a mejorar el entendimiento de un requisito mediante la simulación de la aplicación.
- Una metodología híbrida de desarrollo que tome las ventajas de las metodologías MDWE y ágiles para mejorar el desarrollo de aplicaciones Web.

Aunque ya existe trabajo [9] referido a la integración de metodologías ágiles y dirigidas por modelos, nuestro trabajo [26] fue el primero en mostrar que era posible lograrlo en el ámbito Web. Este trabajo fue el disparador para el desarrollo de nuestro lenguaje de requisitos llamado WebSpec [25] el cual permite las características mencionadas con anterioridad.

En conclusión, **el principal objetivo de investigación** de esta tesis doctoral es el desarrollo de un lenguaje de dominio específico (DSL) que permita la especificación de requisitos Web formalmente. Como consecuencia, las siguientes tareas pueden ser automatizadas ayudando a mejorar el proceso de desarrollo:

- Mejorar el entendimiento de un requisito mediante la simulación de la aplicación Web.
- Automatizar el testing de un requisito con la derivación automática de tests de interacción.
- Semiautomatizar la derivación de la aplicación a diferentes tecnologías no solo en la primera iteración sino también cuando la aplicación evoluciona utilizando control de cambios.

1.3 Resumen del Contenido de la Tesis Doctoral

El objetivo de esta tesis doctoral es atacado primero entendiendo cómo y por qué las aplicaciones son construidas con dos acercamientos diferentes y como estos pueden ser combinados para mejorar su desarrollo. Un punto en el cual los dos acercamientos se quedan cortos es que el testeado manual es una tarea compleja; y como consecuencia nos da el punta pie para el desarrollo de un DSL multipropósito para la especificación de requisitos Web. Como se muestra en los diferentes capítulos de esta tesis, el lenguaje fue originalmente creado para especificar requisitos funcionales pero lo hemos extendido para permitir la validación de modelos de i^* (Capítulo 5) y para expresar requisitos de personalización y accesibilidad (Capítulo 6).

1.3.1 WebTDD

WebTDD es una metodología ágil [26] para el desarrollo de aplicaciones Web; esta basado en ciclos cortos de desarrollo (llamados sprints) que ayudan a obtener feedback rápido de los clientes. Los tests son utilizados para dirigir el proceso de desarrollo y al mismo tiempo verificar que los requisitos son correctamente implementados. Las tecnologías basadas en modelos son utilizadas para desarrollar la aplicación creando y/o actualizando modelos y transformándolos en código. En cada sprint de WebTDD, un conjunto de requisitos es implementado y una nueva versión de la aplicación es entregada al cliente. Es común que los sprints duren 2 semanas y cubran el ciclo completo de desarrollo desde la captura de los requisitos, el desarrollo y el testing.

Al comienzo de cada sprint existe un conjunto de requisitos que necesitan ser implementados. WebTDD define un conjunto de actividades a ser desarrolladas para implementar cada requisito (Fig 1.1):

1. Cada requisito es capturado en Mockups (páginas HTML simples) y diagramas WebSpec (Paso 1 de la Fig. 1.1). Los mockups ayudan a acordar el *look and feel* de la aplicación y los diagramas WebSpec permiten especificar los comportamientos de navegación e interacción. Durante este proceso podemos mejorar la etapa de elicitación de requisitos utilizando la simulación que WebSpec provee. Además, si el control de cambios de WebSpec esta activado, podemos capturar estos cambios para un uso posterior.
2. Luego derivamos en forma automática (Paso 2) un conjunto de tests que la aplicación debe pasar para satisfacer los requisitos capturados. Este proceso es automático y una suite de tests es derivada de cada diagrama.
3. Como en el desarrollo dirigido por tests (TDD [10]) “convencional”, ejecutamos los tests antes de comenzar con la implementación (Paso 3) con el fin de chequear que la aplicación todavía no satisface los requisitos. Los tests que fallen mostraran que caminos de interacción no son satisfechos por la aplicación aún.

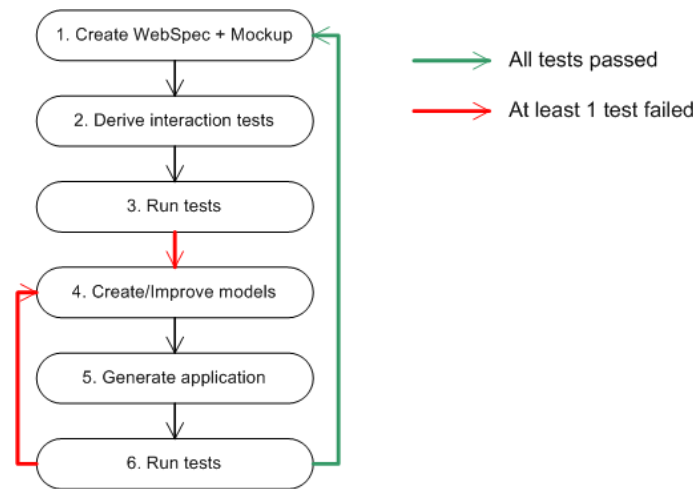


Fig. 1.1. WebTDD

4. Luego las actividades de modelado comienzan (Paso 4); se crea o mejora un conjunto de modelos en la tecnología basada en modelos elegida para el proyecto (por ejemplo WebRatio o MagicUWE). Si habíamos activado el control de cambios de WebSpec, los cambios en los requisitos pueden ser mapeados en forma semi automática en los modelos evitando pérdidas de tiempo.
5. Utilizando la derivación automática a código que la MDWE tool soporta, derivamos una aplicación Web (Paso 5).
6. Finalmente, chequeamos que el requisito haya sido implementado correctamente ejecutando los tests que habíamos derivado previamente (Paso 6). Si al menos un test falla, entonces tenemos que volver modificar los modelos y derivar la aplicación de nuevo hasta que todos los tests pasen. Si todos los tests pasan, podemos comenzar el ciclo nuevamente con el siguiente requisito hasta que no queden más requisitos por ser implementados en el sprint.

Debemos remarcar que WebTDD es independiente de la tecnología basada en modelos que se utilice ya que las diferentes actividades no dependen de los diferentes artefactos o mecanismos de modelado [26].

1.3.2 WebSpec

WebSpec es un lenguaje de dominio específico visual [14] que permite la especificación de requisitos Web de navegación, interacción e interfaz gráfica. El principal artefacto para especificar requisitos es el diagrama WebSpec que puede contener interacciones, navegaciones y comportamientos ricos.

Un diagrama WebSpec define un conjunto de escenarios que la aplicación Web debe satisfacer. Puede contener 2 elementos principales: interacciones y transiciones (que a su vez pueden ser navegaciones o comportamientos ricos). Las interacciones representan puntos donde el usuario puede interactuar con la aplicación y las transiciones representan un movimiento de un punto de interacción a otro. Por lo tanto, un diagrama WebSpec puede ser visto como un grafo donde las interacciones son los nodos y las transiciones representan las aristas. Un escenario es representado como una secuencia de interacciones y transiciones, por ejemplo `<interaction1, navigation1, interaction2, rich1, interaction3>` define un posible camino de interacción entre el usuario y la aplicación Web.

La Fig. 1.2 muestra un diagrama WebSpec para nuestra historia de usuario de ejemplo: “Como cliente, quiero poder buscar productos por su nombre y ver sus detalles”. El diagrama es construido iterativamente entre el cliente y el analista teniendo varias reuniones. Debido a que el uso de WebSpec no está atado a una metodología en particular, podemos utilizar las reuniones de larga duración típicas de métodos unificados o las reuniones cortas típicas de las

metodologías ágiles. La construcción del diagrama puede ser mejorada utilizando mockups y simulando la aplicación (como se muestra en los próximos capítulos); sin embargo, esperamos que con un poco de entrenamiento el cliente sea capaz de construir los diagramas solo. Como ejemplo, en el diagrama de la Fig. 1.2 define los caminos de navegación que el usuario debe seguir desde la página home a la página de resultados y luego a la página de detalle del producto. Además, el usuario debe poder volver atrás en la página de resultados desde la de detalle y también volver a la página home.

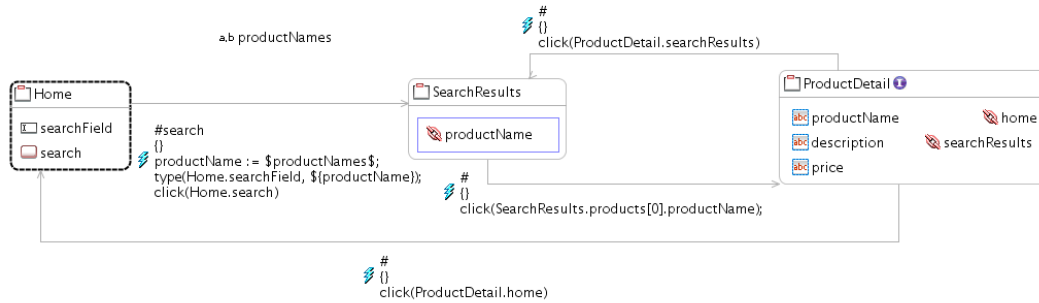


Fig. 1.2. Diagrama Webspec para el escenario de búsquedas por nombre

En un diagrama WebSpec, una *interacción* representa un punto donde el usuario interactúa con la aplicación utilizando sus elementos de interfaz gráfica (widgets). Formalmente, representan el estado de una página Web la cual ha sido cargada inicialmente o cuando ha cambiado como consecuencia de un comportamiento rico. Las interacciones poseen un nombre (único por diagrama) y pueden contener widgets tales como: etiquetas, listas, botones, cajas de selección y paneles. Las etiquetas definen el contenido (información) mostrada por la interacción. Existen 2 tipos de widgets para permitir la composición de estos: ListPanel y Panel. El ListPanel representa una repetición de los elementos que contiene y el panel define un simple contenedor de widgets. Las interacciones están representadas gráficamente con un rectángulo con los bordes redondeados (Fig. 1.3) que contiene el nombre de la interacción y sus widgets. Un diagrama WebSpec debe tener al menos una interacción inicial la cual se encuentra representada con bordes punteados. Para especificar que propiedades debe satisfacer la aplicación hacemos uso de invariantes (expresiones booleanas) en las interacciones del diagrama. Cada interacción (implícitamente o explícitamente) definen un invariante que especifica las propiedades que deben ser satisfechas en los escenarios especificados por el diagrama (en el caso que no se defina uno explícitamente se asumen que el invariante es *true*).

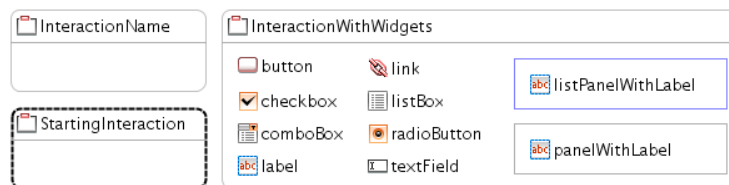


Fig. 1.3. Una interacción en WebSpec

En WebSpec, una navegación esta representada gráficamente (Fig. 1.4) con flechas grises mientras que su nombre, precondition y las acciones que la activan están mostradas como etiquetas sobre ella. En particular, su nombre aparece con el carácter “#” como prefijo, su precondition entre {} y las acciones en las siguientes líneas. Debemos remarcar que la idea detrás de las acciones de una transición (sean estas navegaciones o comportamientos ricos) es

que su ejecución produce una transición entre las interacciones y no al revés. Una transición debe ser entendida como: “Si la precondition se satisface y el usuario ejecuta la secuencia de acciones, la aplicación debe transitar a la interacción destino”.

Una navegación de una interacción a otra puede ser activada si su precondition se satisface, ejecutando su secuencia de acciones tales como: clicar en un botón, agregar texto a un campo, etc. Así como los invariantes, las preconditiones pueden referenciar a variables definidas previamente en el diagrama. Las acciones están escritas de acuerdo a la siguiente sintaxis: $\text{var} := \text{expr} \mid \text{actionName}(\text{arg1}, \dots, \text{argN})$ (una gramática BNF [12] completa puede ser encontrada en el Apéndice A).

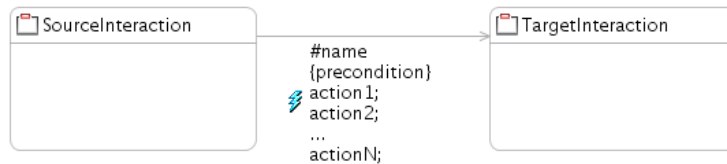


Fig. 1.4. Una navegación en WebSpec

Por otro lado, la aplicación puede cambiar el estado de la UI como consecuencia de algunas acciones ejecutadas por el usuario. Por ejemplo, cuando el mouse esta “sobre” un widget, más información debe ser mostrada en un pop-up o cuando se esta escribiendo texto en un campo de texto, pueden aparecer opciones como en un campo de autocompletado. Estos cambios locales son usuales en las tan llamadas aplicaciones ricas de Internet (RIA [13]) y es común hoy en día que los clientes pidan requisitos de este estilo, tanto explícitamente (“Yo quiero un campo auto-complete”), o implícitamente (“Quiero que la información aparezca como hace Amazon.com”). Estos comportamientos ricos están siendo cada vez más usados en las aplicaciones Web 2.0 pero también en las tradicionales.

En una aplicación Web, un comportamiento rico es percibido como un cambio local en la interfaz que no agrega un elemento nuevo a la navegación del explorador Web. Para especificar un comportamiento rico en WebSpec, utilizamos flechas rojas con líneas punteadas (Fig 1.5) y poseen las mismas propiedades de una navegación (nombre, precondition y acciones).

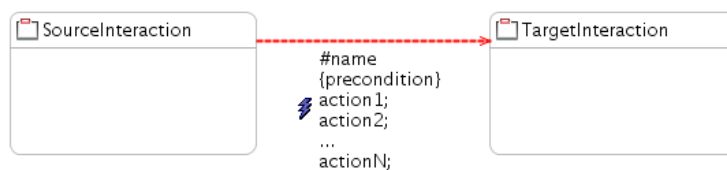


Fig. 1.5. Un comportamiento rico en WebSpec

Mejorando el entendimiento de los requisitos con simulación

Con el objetivo de mejorar la etapa de elicitación de requisitos, los diagramas de WebSpec pueden simular la aplicación en desarrollo. La simulación es importante para reducir la diferencia en el entendimiento de un requisito entre los clientes y los analistas y por lo tanto ayuda a obtener feedback real de ellos. Usualmente, los artefactos para la captura de requisitos [23] requieren un determinado nivel de conocimientos para que los clientes puedan entenderlos por completo, causando problemas de entendimiento durante la etapa de elicitación que luego son controlados cuando la aplicación se encuentra en pleno desarrollo.

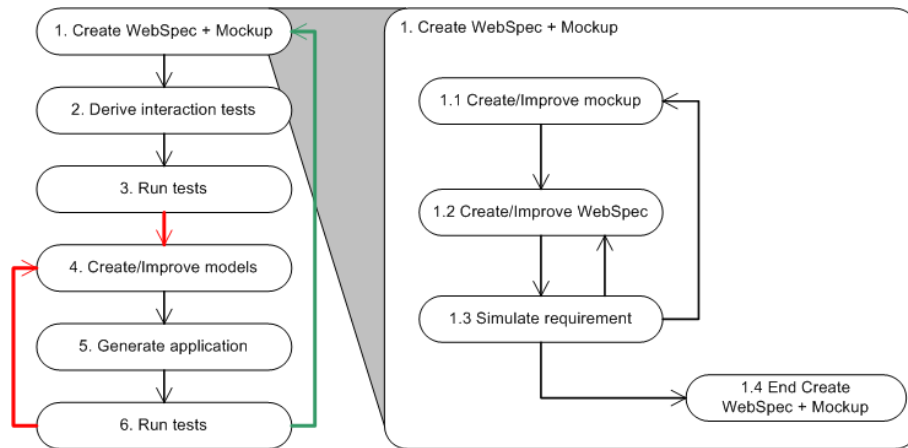


Fig. 1.6. La simulación de WebSpec en el contexto de WebTDD

En WebTDD, la simulación puede ser utilizada cuando creamos los mockups y los diagramas. En la Fig. 1.6 mostramos en detalle la actividad de creación de Mockups y Webspec; comenzamos creando mockups para darle un contexto a los clientes. Luego creamos los diagramas WebSpec de acuerdo a los requisitos de estos y para chequear el comportamiento esperado, simulamos los diferentes caminos de interacción. Una vez que hayamos acordado el requisito, la actividad de creación de Mockups y WebSpec termina.

Para poder dar soporte a la simulación de la aplicación, WebSpec permite la asociación entre las interacciones con los mockups y entre los widgets de WebSpec con sus correspondientes elementos de interfaz en el mockup. Utilizando esta asociación, podemos cambiar entre la especificación de WebSpec con el ejemplo de UI que tenemos en el mockup ayudando a entender el requisito. Los mockups pueden ser creados con herramientas como Balsamiq [2], Axure [1] o HTML plano. Por ejemplo, en la Fig 1.7, mostramos un mockup para la página de detalle de productos creado con Balsamiq. El mockup muestra la información que debe ser mostrada en la página: el nombre del producto, su descripción, precio y los links a la página home y a los resultados. La Fig. 1.8 muestra una simple asociación entre el mockup de la Fig. 1.7 con su correspondiente interacción y widgets.

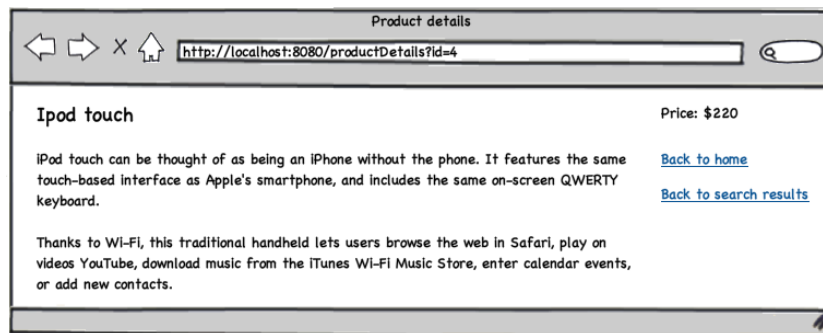


Fig. 1.7. Mockup de Balsamiq para la página de detalle de productos

Nuestra simulación abre un navegador Web con los mockups desarrollados y muestra descripciones y ejecuta acciones que muestran lo que un usuario hipotético de la aplicación haría. Es riguroso porque a diferencia de la simulación provista por herramientas como Balsamiq [2], no solo mostramos las transiciones entre las páginas sino que también ejecutamos acciones reales y proveemos descripciones de cuales serian las salidas reales de la aplicación directamente sobre los mockups. Estas descripciones son generadas automáticamente y son fáciles de comprender ya que están escritas en lenguaje natural. De esta forma para cada diagrama

WebSpec, un conjunto de simulaciones es generado automáticamente el cual puede ser utilizado en cualquier momento por clientes para comprender el significado de un diagrama y proponer cambios o mejoras.

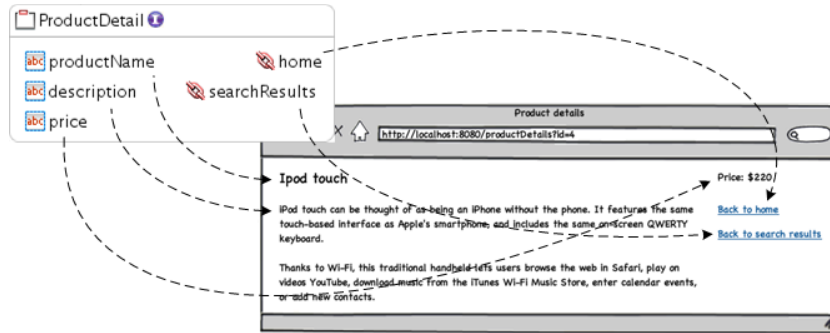


Fig. 1.8. Asociación entre un mockup y una interacción

Validando la implementación de los requisitos con derivación automática de tests

Los requisitos nuevos deben ser validados para garantizar su correcta implementación mientras que los previos siguen funcionando como se espera. Sin embargo, realizar esta tarea eficientemente es una tarea complicada y por lo tanto mantener los requisitos actualizados se vuelve extremadamente importante.

Una manera conocida de validar los requisitos consiste en ejecutar tests automatizados (que expresan los requisitos) sobre la aplicación. Si uno de estos tests falla, entonces un requisito no es satisfecho por la aplicación. En particular, los tests de interacción juegan un papel fundamental en la industria ya que ejecutan un conjunto de acciones de la misma forma que un usuario lo haría en un navegador Web y por lo tanto su uso continua creciendo [21]. En la Fig. 1.9 mostramos en más detalle las actividades ejecutadas durante el ciclo WebTDD; primero necesitamos elegir un conjunto de diagramas WebSpec que expresen el nuevo requisito (Paso 2.1) y de ellos derivar en forma automática un conjunto de tests de interacción (Paso 2.2). Luego, ejecutamos dichos tests eligiendo su correspondiente test suite (Paso 3.1) que ha sido derivada utilizando algún framework de tests (por ejemplo JUnit) (Paso 3.2).

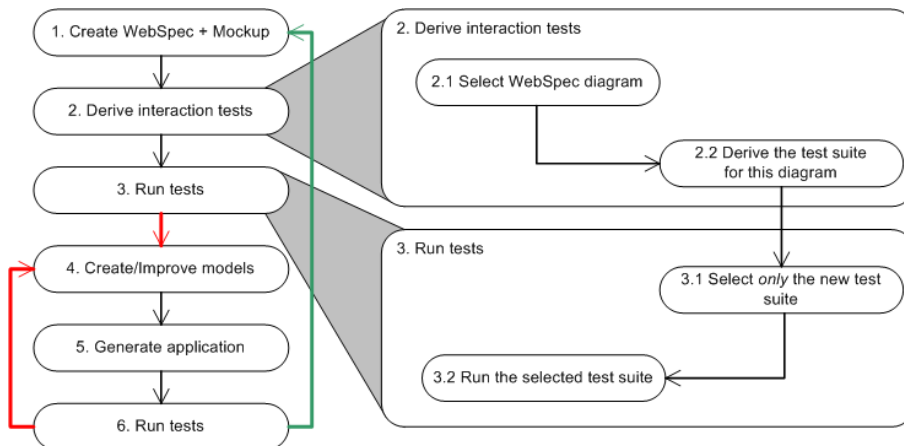


Fig. 1.9. Derivación de tests de WebSpec en el contexto de WebTDD

El proceso de transformación de los diagramas WebSpec en el conjunto de tests es automático y puede ser formalmente descrito en un algoritmo que es aplicado sobre los diagramas (Capítulo 4 y Apéndice A).

El algoritmo sigue los siguientes pasos:

1. Crear la test suite.
2. Computar todos los posibles caminos para el diagrama.
3. Para cada camino:
 - a) Crear un test.
 - b) Abrir la URL de la interacción inicial.
 - c) Agregar todos los pasos en el camino desde la interacción inicial hasta el fin del mismo incluyendo las aserciones e invariantes.

Esta transformación es independiente de la tecnología y puede ser luego utilizada para derivar tests en alguna tecnología en particular (por ejemplo Selenium tests).

Evolución semi automática de la aplicación utilizando el control de cambios de WebSpec

La captura de los cambios en los requisitos es una característica importante para predecir el impacto de estos en la aplicación. Aunque algunos artefactos para la captura de requisitos [17] proveen extensiones para soportar el control de cambios, en el campo de la ingeniería Web estos aspectos han sido ignorados (revisar Capítulo 4 y Apéndice A para más detalles).

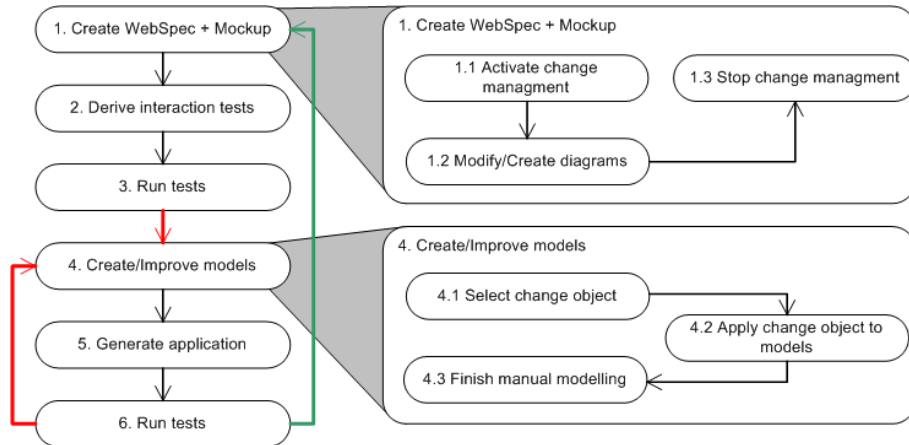


Fig. 1.10. Derivación de la aplicación en el contexto de WebTDD

En WebSpec, los cambios son grabados en objetos de cambio (change objects) que agrupan un conjunto de cambios en los diagramas. Los change objects son creados aun en las fases iniciales (cuando el diagrama esta siendo creado). Los diagramas WebSpec pueden tener diferentes cambios de grano grueso, como el agregado o borrado de una interacción o transición. Estos elementos también pueden ser modificados por el agregado o borrado de widgets a una interacción, cambio en los invariantes, etc. Respecto a las transiciones, podemos agregar o modificar sus precondiciones, cambiar su origen y destino, o las acciones que las activan. Cuando un usuario modifica un diagrama, un objeto de cambio es creado y la secuencia de cambios es grabada como instancias en un metamodelo (Capítulo 4 y Apéndice A). En la Fig 1.10 se muestra las actividades en el contexto de WebTDD; cuando estamos creando o modificando diagramas, activamos el control de cambios de WebSpec para grabar dichos cambios. Luego, cuando comenzamos con las tareas de modelado podemos aplicar estos cambios en forma semi automática a nuestros modelos para mejorarlos. Como WebSpec no soporta todos los tipos de

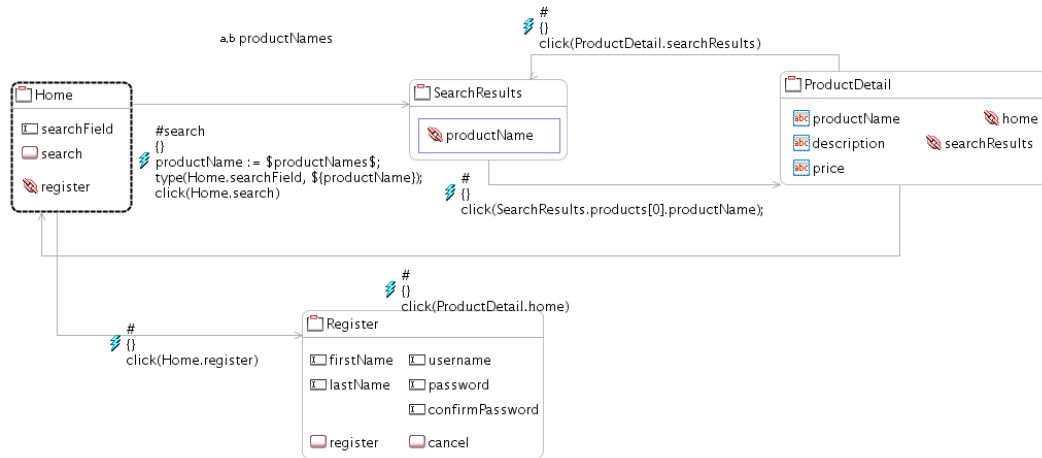


Fig. 1.11. Extensión del diagrama de búsqueda con una interacción de registro

cambios (especialmente aquellos relacionados con como fue modelada la aplicación) debemos continuar con las tareas de modelado en forma manual.

Como ejemplo, supongamos que agregamos una interacción de registro (Register) con sus widgets y un link desde la interacción Home (Fig. 1.11). Este cambio en el diagrama genera un nuevo change object que contiene los siguientes elementos: una nueva interacción (Register), una nueva navegación (Home → Register), un nuevo link (register) en la interacción Home y un nuevo conjunto de widgets en la interacción Register.

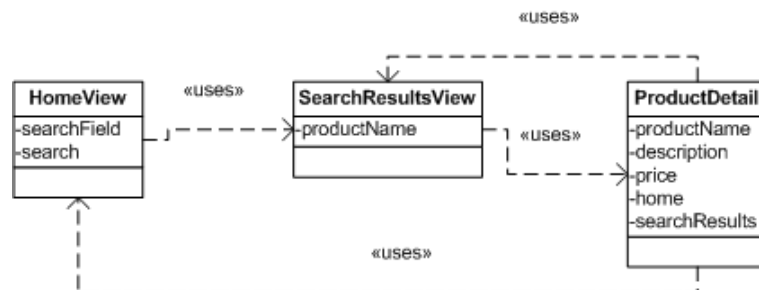


Fig. 1.12. Versión existente del modelo de UI antes de que el objeto de cambio haya sido aplicado

Asumiendo que estamos modelando la UI con un modelo de clases (Fig 1.12), lo podemos actualizar en forma automática al mostrado en la Fig 1.13 utilizando el control de cambios provisto por WebSpec (Capítulo 4 y Apéndice A).

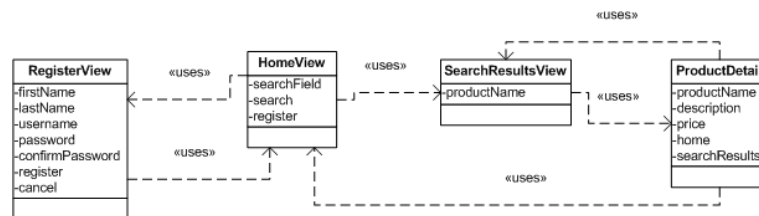


Fig. 1.13. Versión modificada del modelo de UI luego de haber aplicado el change object

1.3.3 Implementación

Una tool para WebSpec ha sido implementada como plugin de Eclipse utilizando tecnologías EMF [3] y GMF [4] y esta disponible actualmente como proyecto open source¹.

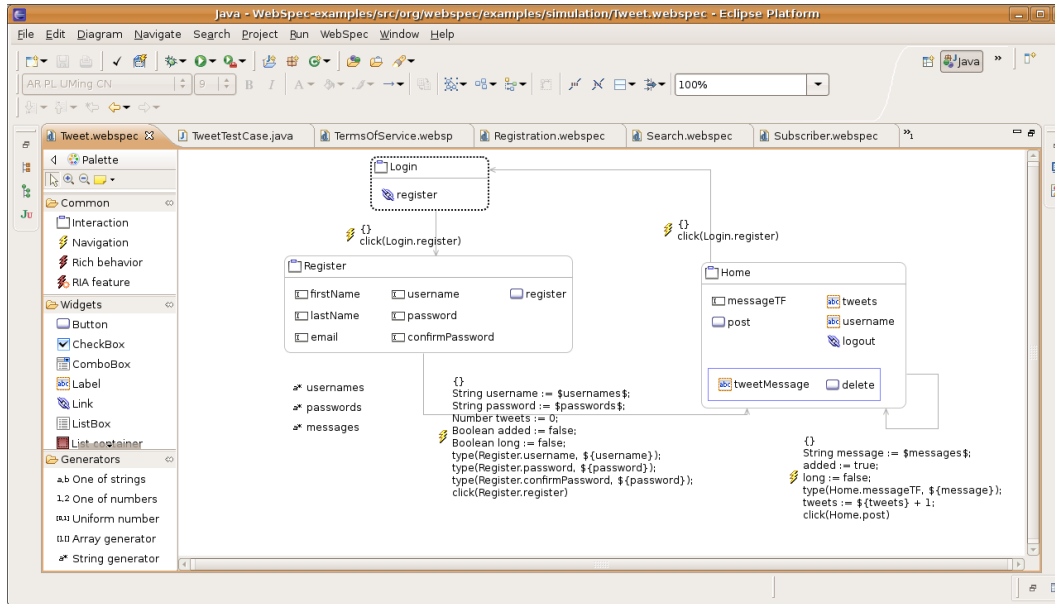


Fig. 1.14. Plugin de Eclipse para WebSpec

El plugin soporta las siguientes características:

- Creación de diagramas WebSpec: un editor visual permite la creación, modificación y actualización de los diagramas. Las propiedades de los elementos pueden ser modificadas seleccionando cada elemento y actualizando los editores de propiedades en la vista de propiedades.
- Asociación con mockups HTML: tomando como ventaja el framework de Eclipse, los mockups HTML son archivos dentro del proyecto. El editor permite seleccionar una interacción y su mockup HTML fácilmente. La asociación entre los widgets es realizada editando la propiedad location del widget WebSpec en la vista de propiedades.
- Simulación de la aplicación: utilizando la asociación previa, el plugin abre los mockups en un navegador Web y muestra descripciones de cual sería el comportamiento esperado. Esta característica ha sido implementada extendiendo el mecanismo de comunicación de Selenium [6] y utilizando un plugin de JQuery [5] para mostrar las descripciones.
- Derivación de tests a selenium: Como mostramos anteriormente, cada diagrama es transformado en un modelo de tests. Luego el plugin permite la traducción del modelo de tests a tests de Selenium.
- Manejo de cambios: Utilizando el patrón observer [15] de EMF, nos registramos para recibir notificaciones de los cambios en el diagrama y así el plugin crea el modelo de cambios. El usuario del plugin es quien decide si comenzar a grabar los cambios o no. Cuando algunos cambios han sido capturados y el usuario detiene el grabado, el modelo de cambios es grabado en un archivo para ser usado luego.
- Generación/Actualización de clases GWT y Seaside: Utilizando el modelo de cambios grabado con anterioridad, el modelo de UI puede ser generado automáticamente. Actualmente, el plugin soporta la generación de clases GWT y Seaside y maneja no solo la primera versión de los cambios sino cambios incrementales.

¹ Mirar <http://code.google.com/p/webspec-language/> para más detalles

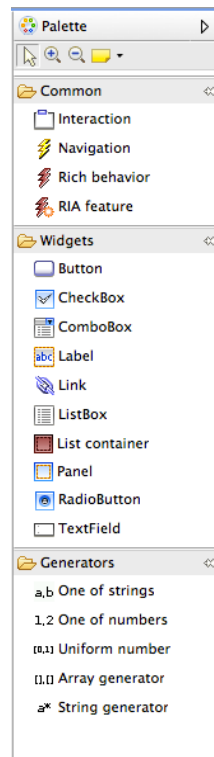


Fig. 1.15. La paleta de WebSpec

La Fig. 1.14 muestra una pantalla del plugin de Eclipse. En la Fig. 1.15 podemos ver más detalles de la paleta de WebSpec que permite la creación de cada elemento WebSpec realizando un drag and drop de cada elemento sobre el diagrama. Luego, si seleccionamos un elemento, podemos editar sus propiedades en la vista de propiedades de Eclipse (Fig. 1.16). En las próximas subsecciones daremos más detalles de como se han implementado cada una de las características presentadas por el plugin.

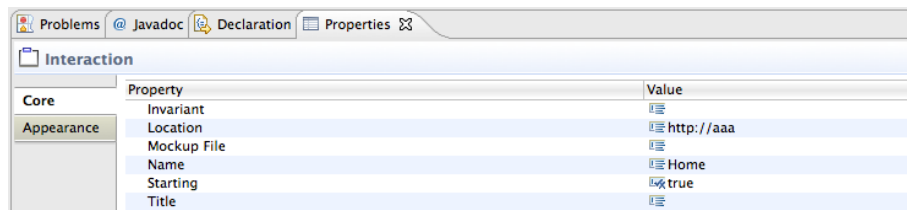


Fig. 1.16. Las propiedades de un diagrama WebSpec

Simulación

La característica de simulación implica la implementación de 3 elementos: la transformación entre WebSpec y los modelos de simulación, la asociación entre los mockups y la ejecución de la simulación. La transformación entre WebSpec y los modelos de simulación ha sido implementada directamente en Java debido a que es mucho más sencillo de realizar los algoritmos de cómputo de caminos en este lenguaje que en QVT. Para realizar la transformación simplemente abrimos el menú de WebSpec (Fig. 1.17) y elegimos el item Simulate.

La asociación con los Mockups has sido implementada fácilmente tomando como ventaja el ambiente de Eclipse. Agregamos una nueva propiedad para las interacciones y widgets que abre un diálogo para elegir un archivo el cual permite elegir el mockup HTML.

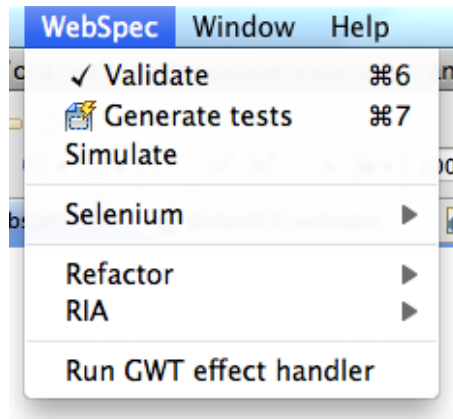


Fig. 1.17. El menú de WebSpec

La simulación en si fue la parte más compleja de implementar y requirió la extensión del framework Selenium. Utilizamos el mecanismo de comunicación existente de Selenium para abrir el navegador Web y ejecutar las acciones en el mismo. Como se muestra en la Fig. 1.18, mostramos descripciones sobre los mockups utilizando un plugin de JQuery. Para hacerlo funcionar, tuvimos que extender el framework de Selenium para que se cargue dichas librerías y mostrar las descripciones cuando sea necesario. Debemos notar que el mismo mockup (el cual puede ser más rico que una interacción debido a que posee más widgets) puede ser utilizado en múltiples y diferentes simulaciones. Nuestro acercamiento mantiene el mockup en el mismo estado en el cual fue construido sin quitar ninguno de los widgets existentes debido a que ellos confundirían a los clientes acerca de la presencia o ausencia de los mismos.

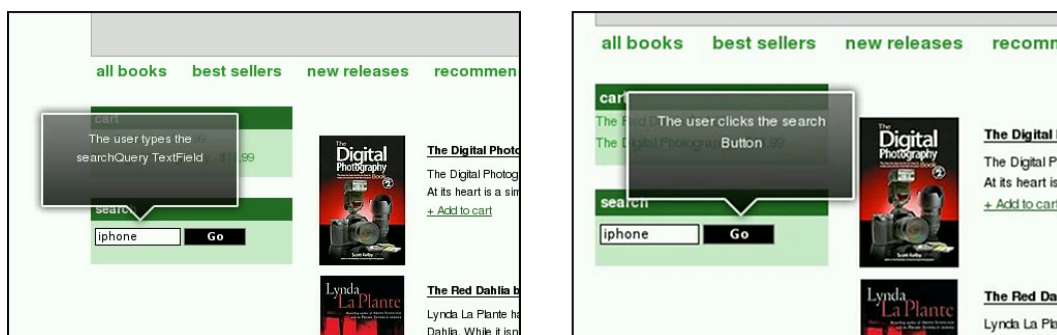


Fig. 1.18. Ejemplo de simulación en WebSpec

Validación de requisitos

Para permitir la validación de requisitos tuvimos que realizar 2 tareas: transformar a los diagramas en modelos de tests y luego derivar estos tests a una tecnología en particular. La transformación entre los modelos ha sido implementada aprovechando la arquitectura y transformaciones existentes para la simulación debido a que ambas utilizan algoritmos de cómputo de caminos.

Con el fin de transformar los modelos de tests a unos dependientes de la tecnología, utilizamos transformaciones de modelos a texto. Actualmente, el plugin soporta la derivación de tests a Selenium y estamos trabajando en la derivación a Webdriver [7]. Como ejemplo mostramos a continuación el código generado para nuestro caso de ejemplo en el framework Selenium:

```
(01) selenium.open("http://localhost:8080/index.html");
(02) selenium.type("id=searchField", "Ipod");
(03) selenium.click("id=search");
(04) selenium.waitForPageToLoad("30000");
(05) selenium.click("id=product0");
(06) selenium.waitForPageToLoad("30000");
(07) assertTrue(selenium.getText("id=productName").equals("Ipod"));
(08) selenium.click("id=home");
(09) selenium.waitForPageToLoad("30000");
(10) selenium.type("id=searchField", "book");
(11) selenium.click("id=search");
(12) selenium.waitForPageToLoad("30000");
(13) selenium.click("id=product0");
(14) selenium.waitForPageToLoad("30000");
(15) assertTrue(selenium.getText("id=productName").equals("book"));
(16) selenium.click("id=home");
```

La línea 1 abre la aplicación en el navegador Web. Las líneas 02-04 buscan al producto Ipod. Las líneas 05-06 eligen el primer producto que fue obtenido como resultado y finalmente la línea 07 asegura que el producto elegido tiene el nombre Ipod. Las Líneas 08-09 navegan a la página Home. Las líneas 10-12 buscan al producto book, las líneas 13-14 seleccionan el primer producto que fue obtenido como resultado y finalmente la línea 15 asegura que el producto seleccionado tiene como nombre book. La línea 16 navega a la página Home.

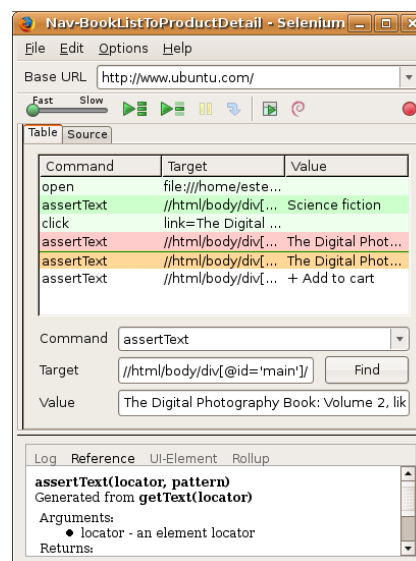


Fig. 1.19. Un test que falla

Como ejemplo, los tests de Selenium pueden ser ejecutados en el Selenium IDE, la Fig 1.19 muestra un test que falla exponiendo que un requisito no ha sido correctamente implementado en la aplicación aún.

Cambios en los requisitos

Cuando un diagrama es modificado, grabamos sus cambios en un archivo de cambios. Este archivo no es más que una serialización del modelo de cambios en formato XML. Para capturar los cambios utilizamos el patrón observer e incrementalmente lo vamos construyendo; luego lo serializamos a XML. Los archivos de cambios son leídos y utilizamos para actualizar la aplicación utilizando manejadores de cambios (un componente que se encarga de mapear los cambios en WebSpec en los de una tecnología en particular). El plugin soporta la generación de clases y métodos compatibles con Seaside y GWT y estamos trabajando activamente para derivar a modelos de WebRatio [8].

Como ejemplo del uso de manejadores de cambios, mostramos a continuación el uso de objetos de cambio en nuestro ejemplo de actualización (el agregado de la registración que hemos mostrado previamente) en GWT. Con el fin de mantener la discusión acotada mostramos la clase `RegisterView` creada por el manejador de cambios de GWT.

Básicamente, las líneas 09-15 definen las variables de instancia que representan a los widgets y las líneas 21-29 inicializan estos objetos con sus respectivas clases GWT. Debemos notar que la clase `RegisterView` extiende de `VerticalPanel` (una clase base en GWT para la implementación de UIs).

```
(01) package org.webspeclanguage.re;
(02)
(03) import com.google.gwt.user.client.ui.VerticalPanel;
(04) import com.google.gwt.user.client.ui.TextBox;
(05) import com.google.gwt.user.client.ui.Button;
(06)
(07) public class RegisterView extends VerticalPanel {
(08)
(09)     private TextBox firstName;
(10)     private TextBox lastName;
(11)     private TextBox username;
(12)     private TextBox password;
(13)     private TextBox confirmPassword;
(14)     private Button register;
(15)     private Button cancel;
(16)
(17)     public RegisterView() {
(18)         this.initializeComponent();
(19)     }
(20)
(21)     public void initializeComponent() {
(22)         this.firstName = new TextBox();
(23)         this.lastName = new TextBox();
(24)         this.username = new TextBox();
(25)         this.password = new TextBox();
(26)         this.confirmPassword = new TextBox();
(27)         this.register = new Button();
(28)         this.cancel = new Button();
(29)     }
(30) }
```

En la Fig. 1.20 mostramos una representación visual de la clase `RegisterView` donde hemos aplicado un poco de estilo para mejorar el *look and feel* de la misma.

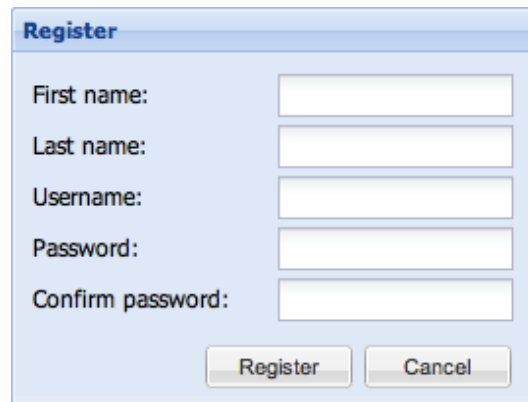
A screenshot of a web registration form titled "Register". The form has a light blue header bar with the title. Below the header, there are five text input fields arranged vertically, each preceded by a label: "First name:", "Last name:", "Username:", "Password:", and "Confirm password:". At the bottom of the form, there are two buttons: "Register" and "Cancel". The form is enclosed in a thin blue border.

Fig. 1.20. Una representación visual de la clase RegisterView

1.4 Conclusiones

El desarrollo de aplicaciones Web es un proceso complejo y que consume mucho tiempo. A su vez involucra a equipos de desarrollo multidisciplinarios con diferentes conocimientos y roles. Para estos equipos es común enfrentar el desafío de hacer evolucionar aplicaciones Web en períodos de tiempo corto con el fin de satisfacer los nuevos requisitos del mercado. Principalmente porque actualizar una aplicación de acuerdo a los nuevos requisitos es una tarea complicada si queremos evitar el problema de romper la funcionalidad existente.

En la literatura, la solución de facto han sido las metodologías MDWE que usan modelos para desarrollar la aplicación. Sin embargo, estas son más pesadas que sus pares ágiles y el feedback de los clientes es obtenido muy tarde. Por otro lado las metodologías ágiles están centradas en el código y requieren un montón de esfuerzo manual para varias tareas incluidas el testeo de la misma aplicación.

Para resolver estos problemas hemos presentado en esta tesis doctoral una metodología híbrida llamada WebTDD que mezcla las ventajas de las metodologías MDWE con las ágiles. Esta metodología ha sido el disparador para el desarrollo del principal componente de esta tesis; un DSL para la especificación de requisitos Web llamado WebSpec. Hemos mostrado como especificar requisitos Web usando el lenguaje y al mismo tiempo simular la aplicación en desarrollo. La simulación es soportada mediante el uso de mockups y esta ayuda a mejorar el entendimiento del requisito por parte de los diferentes miembros del equipo. Como hemos dicho anteriormente, el testing es crucial en este contexto y hemos mostrado como un conjunto de tests han sido derivados de cada diagrama WebSpec permitiendo validar si el requisito ha sido implementado correctamente o no. Finalmente, tomando ventaja del sistema de cambios que WebSpec posee, hemos mostrado como actualizar la aplicación en forma semi automática.

Cabe mencionar que WebTDD es la primera metodología híbrida en mostrar que la combinación de métodos ágiles como métodos basados en modelos es posible en el ámbito Web. Además, WebSpec es el primer DSL para la especificación de requisitos Web que permite las características mencionadas con anterioridad y es independiente del proceso de desarrollo. En esta tesis hemos utilizado a WebTDD porque es un *matching* perfecto para las características de WebSpec.

References

1. Axure - wireframes, prototypes, specifications. available at: <http://www.axure.com/>.
2. Balsamiq. available at: <http://www.balsamiq.com/products/mockups>.
3. Eclipse emf. available at: <http://www.eclipse.org/modeling/emf/>.
4. Eclipse gmf. available at: <http://www.eclipse.org/modeling/gmp/>.
5. jquery: The write less, do more, javascript library. available at: <http://jquery.com/>.
6. Selenium web application testing framework. <http://seleniumhq.org/>.

7. Webdriver. available at: <http://webdriver.googlecode.com>.
8. The webratio tool suite. available at: <http://www.webratio.com>.
9. S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
10. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
11. S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks and Isdn Systems*, 33:137–157, 2000.
12. N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, Sept. 1956.
13. J. Duhl. Rich internet applications. a white paper sponsored by macromedia and intel, idc report, 2003.
14. M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
16. J. Gómez and C. Cachero. *OO-H Method: extending UML to model web interfaces*, pages 144–173. IGI Publishing, Hershey, PA, USA, 2003.
17. I. Jacobson. *Object-Oriented Software Engineering: a Use Case driven Approach*. Addison-Wesley, Wokingham, England, 1995.
18. N. Koch, A. Knapp, G. Zhang, and H. Baumeister. *UML-BASED WEB ENGINEERING - An approach based on standards*, chapter 7, pages 157–191. Springer, 2008.
19. D. Lowe. Web system requirements: an overview. *Requir. Eng.*, 8(2):102–113, 2003.
20. R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
21. E. M. Maximilien and L. Williams. Assessing test-driven development at ibm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society.
22. A. McDonald and R. Welland. Web engineering in practice, 2001.
23. D. L. Moody. The éphysicse; of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Software Eng.*, 35(6):756–779, 2009.
24. O. Pastor, S. M. Abrahão, and J. Fons. An object-oriented approach to automate web applications development. In *Proceedings of the Second International Conference on Electronic Commerce and Web Technologies*, EC-Web 2001, pages 16–28, London, UK, 2001. Springer-Verlag.
25. E. Robles Luna, I. Garrigós, J. Grigera, and M. Winckler. Capture and evolution of web requirements using webspec. In *Proceedings of the 10th international conference on Web engineering*, ICWE'10, pages 173–188, Berlin, Heidelberg, 2010. Springer-Verlag.
26. E. Robles Luna, J. Grigera, and G. Rossi. Bridging test and model-driven approaches in web engineering. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Web Engineering*, volume 5648 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02818-210.
27. G. Rossi and D. Schwabe. Modeling and implementing web applications with oohdm. In G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, editors, *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pages 109–155. Springer London, 2008. 10.1007/978-1-84628-923-16.
28. M. Wimmer, A. Schauerhuber, and H. Kargl. On the integration of web modeling languages: Preliminary results and future challenges.

Summary in English

This PhD Thesis is composed of a set of published and submitted papers. Therefore, this chapter is devoted to a description of initial hypotheses, research objectives, and the collection of works that are part of this Thesis, thus justifying its coherence. It should be underlined that this chapter summarizes the scientific content of this PhD Thesis, including research results and final conclusions. Finally, the previous chapter has been written in Spanish, and then translated into English as follows.

2.1 PhD Thesis as a Collection of Papers

In order to write this PhD Thesis as a collection of papers in the University of Alicante, a set of requirements must be followed. These requirements were defined by “*Pleno de la Comisión de Doctorado de la Universidad de Alicante*” on the 2nd of March, 2005; those related to the content of the PhD Thesis are presented as follows:

1. “*The PhD Thesis must include a summary written in one of the two official languages of this region. It should contain objectives, hypotheses and works to justify the coherence of the research.*”
2. “*This summary must include an abstract to present the results, a discussion of them and the final conclusions. This summary must give an idea of the overall content of the PhD Thesis.*”
3. “*The works presented in this PhD Thesis must be published or accepted for publication after the beginning of the PhD. Papers under review can be included in the appendices of the PhD Thesis.*”

In order to fulfil the aforementioned requirements, this PhD Thesis is structured in three parts. Part I consists of a summary in Spanish (Chapter 1) and its corresponding summary in English (Chapter 2). Part II presents a collection of published papers that are the main content of this PhD thesis. Part III presents an article under revision.

2.1.1 Publications Included in this PhD Thesis

A set of the published research papers have been selected to be part of this PhD Thesis due to (i) their scientific contribution and (ii) their relevance. They are described in this section.

Chapter 3

Robles Luna E., Grigera J., Rossi G. *Bridging Test and Model Driven Approaches in Web Engineering. Proceedings of 9th International Conference on Web Engineering (ICWE 2009). 2009. San Sebastian, Spain. Acceptance rate: 24%. Core C*

Robles Luna E., Panach J.I., Grigera J., Rossi G., Pastor O. *Incorporating Usability Requirements in a Test/Model-Driven Web Engineering Approach. Journal of Web Engineering (JWE).* 2010. Impact factor: 0.531. JCR.

This work describes a methodology for Web application development in which tests play a fundamental role driving the development process. Failing tests indicate that part of the required functionality has not been implemented (similar to test driven development [10]). However, different from test driven approaches in which the main development artefact is the code in our methodology we use a model based development in which models abstract but **not** drive the development.

The methodology presented in this chapter defines a good framework for WebSpec (the main contribution of this PhD thesis) being used. In particular, because these methodologies don't provide automatic translation from requirements to tests.

Chapter 4

Robles Luna E., Garrigos I., Grigera J., Winckler M. *Capture and Evolution of Web requirements using WebSpec. Proceedings of 10th International Conference on Web Engineering (ICWE 2010).* Vienna, Austria. Acceptance rate: 20%. Core C

In this chapter we present the domain specific language which is the core part of this PhD thesis. We show its definition and use in the different activities of the development cycle. Though Webspec was initially conceive to be used with the approach presented in chapter 3, we show how it can be used with a unified methodology.

Chapter 5

Robles Luna E., Garrigos I., Mazon J-N., Trujillo J., Rossi G. *An i*-based Approach for Modeling and Tesing Web Requirements. Journal of Web Engineering (JWE).* 2010. Impact factor: 0.531. JCR.

Some development methodologies use an early phase of requirements in which objectives and tasks of the system/organization are defined before more capturing detailed requirements (like the one done with WebSpec). Several times a formal language like i* is used to describe these relationships. In this chapter we show how to use WebSpec with i* to model Web requirements. When we used both artefacts we can semi automatically validate that the objectives described in the i* model are correctly implemented in the application by using the automatic derivation of tests that WebSpec provides.

Chapter 6

Medina, N. M., Burella, J., Rossi G., Grigera J., **Robles Luna E.** *An Incremental Approach for Building Accessible and Usable Web Applications. Proceedings of the 11th International Conference on Web Information System Engineering (WISE 2010).* Hong Kong, China. Acceptance rate: 18.8%. Core A

Robles Luna E., Garrigos I., Rossi G. *Capturing and Validating Personalization Requirements in Web Applications. Proceedings of the 1st Workshop on The Web and Requirements Engineering (WeRE 2010).* Sydney, Australia.

In this chapter we show how to use WebSpec for the specification of non functional requirements like accessibility and personalization of Web application. In each case we provide small extensions to the core language with the intent of allowing the specification of these requirements in the context of WebTDD.

Chapter 7

Burella J., Rossi G., **Robles Luna E.**, Grigera J. *Dealing with Navigation and Interaction Requirement Changes in a TDD-Based Web Engineering Approach. Proceedings of the 11th International Conference on Agile Software Development (XP 2010)*, Springer Verlag, LNCS, 2010. Trondheim, Norway. Core B.

Robles Luna E., Burella J., Grigera J., Rossi G. *A Flexible Tool Suite for Change-Aware Test-Driven Development of Web Applications. Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE 2010)*. 2010. Cape Town, South Africa. Core A.

In this chapter we present WebSpec's change management which allows detecting the code pieces affected by a requirement change. To allow this feature, we establish an association between the changes that happen at the requirement level with those in the implementation. In addition, we present a demonstration of WebSpec's Eclipse tool which gives support for each of the language's features.

Appendix A

Robles Luna E., Rossi G., Garrigos I. *WebSpec: a Visual Language for Specifying Interaction and Navigation Requirements in Web Applications. Requirements Engineering Journal*. In press. Impact factor: 0.931. JCR.

In this chapter we present an evolution of the core language presented in chapter 4 in which we detail the specification of requirements for rich internet applications. In addition, we present details about the language's grammar and an extension of the case of study.

2.1.2 Other publications in International conferences

During the development of this PhD thesis we have published other articles which have not been explicitly included in this document. However, these papers are part of the research done during my PhD studies and complement this PhD thesis:

- **Robles Luna E.**, Escalona M.J., Rossi G. A requirements metamodel for Rich Internet applications. Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFTE 2010). Athens, Greece. Acceptance rate: 9%. Core B.
- Rivero J.M., Rossi G., Grigera J., Burella J., **Robles Luna E.**, Gordillo S. From mockups to user interface models: An extensible model driven approach. Proceedings of the 6th Model-Driven Web Engineering Workshop. (MDWE 2010). Vienna, Austria.
- **Robles Luna E.**, Rossi G., Burella J., Grigera J. Incremental Usability Improvement in an Agile Approach for Web Applications. Proceedings of the 1st workshop Dealing with Usability in an Agile Domain, XP'2010 workshop. (Usability&Agile 2010), 2010. Trondheim, Norway.
- **Robles Luna E.**, Grigera J., Rossi G., Panach J. I. and Pastor O. Introducing Usability Requirements in a Test/Model-Driven Web Engineering Method. Proceedings of 8th International Workshop on Web-Oriented Software Technologies (IWOOST 2009). 2009. San Sebastian, Spain.

2.2 Research Objectives and Initial Hypotheses

Web application development is a complex and time consuming process that involves different stakeholders. It is typical that development teams are usually multidisciplinary (including customers, visual designers, developers, QA staff, etc) and therefore the understanding of the application varies between the team members. In addition, these applications have some unique characteristics like navigational access to information and sophisticated interaction features making their development different from traditional desktop application. As a consequence we can find in the literature different development approaches to build them. However, there are two distinctive groups for web application development: model driven web engineering (MDWE) and agile methodologies.

On one hand, several MDWE approaches have been proposed during the last 20 years [11, 16, 18, 24, 27]. All of them share a common top-down style [28], constructing the web application by describing a set of models at different abstraction levels:

- Content (or Application) Model: defines domain objects and their relationships.
- Hypertext (or Navigation) Model: defines navigation nodes and links that publish information specified by objects in Content Model.
- Presentation Model: refines the Hypertext Model with concrete user interface presentation features like pages, concrete widgets, layout, etc.

The process used in these methodologies is generally top-down, delivering a final web Application in a specific technology using automatic model transformations.

On the other hand, agile methodologies promote early and constant interaction with customers. In this way they can assert that the software built complies with their requirements by constantly delivering prototypes which are developed in short periods of time. Agile approaches argue that software specifications must emerge naturally, enhancing former prototypes along the development until the final application is obtained.

To summarize, while MDWE methodologies facilitate software specification portability, abstraction and productivity, they fail in providing agile interaction with customers because concrete results are obtained too late. On the other hand, while this feature is clearly provided by agile methodologies, they are heavily based on direct implementation and thus fail to provide abstraction, portability and productivity through automatic code generation.

According to several studies [22, 19] in industrial cases, the requirements phase is one of the most important phases of any web development approach. Unfortunately, in the context of MDWE, requirements are generally captured using Use cases [17] or a modified version of them while in agile approaches there is a tendency to replace Use cases with User stories [20]. Regarding the expressive power of both artefacts, they are very **poor to express** the particularities of the Web (e.g. their interactive and navigation-driven nature). In addition, the fast evolution of Web applications (within few weeks) poses additional constraints to allow continuous and timely application **testing** against the requirement specification [19] mainly to validate that new requirements have been correctly implemented without “breaking” previous ones. In this context, capturing and modeling requirements should be efficient enough to accomplish the time constraint. Therefore, it is important that requirements need to be **clearly understood** to provide efficient application evolution.

Taking into account these considerations, **the hypothesis of this PhD thesis** is the improvement of web application development by:

- A formal requirements specification language that automates the requirements validation phase, semiautomates the derivation of the application and helps to understand the requirement through web application simulation.
- An hybrid development approach which takes the advantages of MDWE and agile methodologies to improve web application development.

Though existing work [9] propose the idea of combining agile methodologies with Model driven development our work [26] was the first to show that the approach was feasible in the

Web context. This work was the trigger for the development of our requirement language called WebSpec [25] that allows the aforementioned automatic features.

In conclusion, **the main research objective** of this PhD thesis is the development of a domain specific language (DSL) to allow specifying web requirements formally. As a consequence the following tasks could be automated and help to improve the development process:

- Improve the understanding of the requirement through web application simulation.
- Automate the testing of the requirement with the automatic derivation of interaction tests.
- Semiautomate the derivation of the application to different technologies not only in the first iteration but also when the application evolves using change management support.

2.3 PhD Thesis in a Nutshell

The objective of this PhD thesis is tackled by first understanding how and why applications are built using two different approaches and how they could be combined to improve web application development. One point that both approaches clearly fall short is that manual testing is a hard task; therefore it motivates the development of a multipurpose DSL to specify web requirements. As shown in the different chapters of this thesis, the language was originally created for specifying functional requirements but we have extended it to validate i^* models (Chapter 5) and to allow expressing personalization and accessibility requirements (Chapter 6).

2.3.1 WebTDD

WebTDD is an agile approach [26] for developing web applications; it is based on short development cycles (called sprints) that helps to gather quick feedback from customers. Tests are heavy used to drive the development approach while validating that requirements are correctly implemented. Model based technologies are used to develop the application by creating/updating models and transforming into code. In each sprint of WebTDD, a set of requirements is implemented and a new version of the application is delivered to the customer. It is typical that sprints last 2 weeks and cover the full development cycle from requirements capture, development and testing.

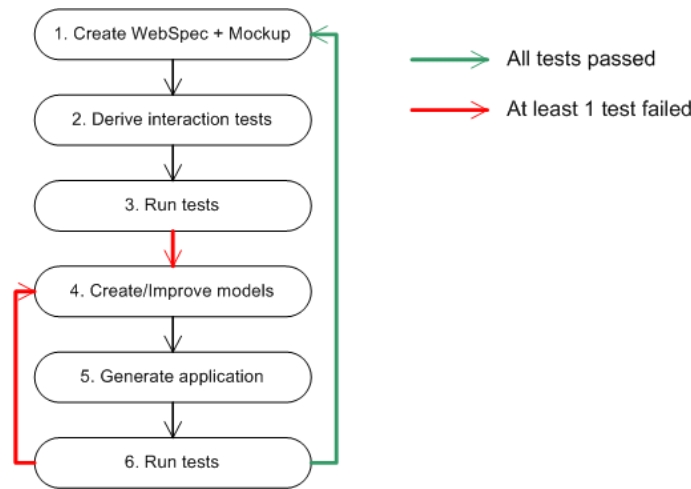


Fig. 2.1. WebTDD

At the beginning of each sprint, there are a set of requirements that need to be implemented. WebTDD poses a set of activities to be performed to implement each requirement (Fig 2.1):

1. Each requirement is captured with mockups (stub HTML pages) and WebSpec diagrams (Step 1 of Fig. 2.1). Mockups help to agree in the look and feel of the application, and WebSpec diagrams capture navigation and interaction behaviours. During this process we can improve the elicitation of requirements using WebSpec simulation. Also, if WebSpec's change management support is activated, we can capture the changes in the diagrams for later use.
2. Next we automatically derive (Step 2) a set of meaningful tests that the application must pass to satisfy the captured requirements directly from WebSpec diagrams. This process is automatic and a test suite is derived from each diagram.
3. As in "conventional" test driven development (TDD [10]), we run them prior to the implementation (Step 3) in order to check that the application does not satisfy the requirements yet. The failing tests will show which are the interaction paths that the application does not satisfy yet.
4. Afterwards, the modelling activities begin (Step 4); we create or enhance a set of models in the model based technology chosen for the project (e.g. WebRatio or MagicUWE). If we have activated the change management support that WebSpec provides, the changes in the requirements could be mapped semiautomatically in the models avoiding wastes of time.
5. Using the automatic model transformation that the MDWE tool supports we derive the web application (Step 5).
6. Finally, we check whether the requirement has been successfully implemented by running the previous tests (Step 6). If at least one test fails, we have to go back, tweak the models and derive the application again until all tests pass. If all tests pass then we can start the process again with the next requirement until we run out of requirements to be implemented in the sprint.

We must notice that WebTDD is independent of the model based technology used for the modelling activities as the different activities does not depend on the specific modelling artefacts or mechanics [26].

2.3.2 WebSpec

WebSpec is a visual domain specific language [14] that allows specifying navigation, interaction and UI Web requirements. The main artefact for specifying requirements is the WebSpec diagram which can contain interactions, navigations and rich behaviours.

A WebSpec diagram defines a set of scenarios that the Web application must satisfy. It can contain two main elements: interactions and transitions (which can be in turn navigations or rich behaviours). Interactions represent points where the user can interact with the application and transitions represent a movement from one point of interaction to another. Therefore, a WebSpec diagram could be seen as a graph where interactions are the nodes of the graph and transitions represent the edges. A scenario is represented by a sequence of interactions and transitions, e.g. <interaction1, navigation1, interaction2, rich1, interaction3> that defines a possible path of interactions between the user and the Web application.

Fig. 2.2 shows a WebSpec diagram for our exemplar user story: "As a customer, I would like to search products by name and see its details". The diagram is constructed iteratively between the customer and the analyst by having several meetings. Since the use of WebSpec is not tight to any particular development process, we can use the long duration meetings of unified development approaches or short meetings where customers are really involved and are typical in agile development. Diagrams' construction could be improved by using mockups and simulating the application (as shown next); however, we expect that with some training the customer would be able to solely build a diagram. As an example, the diagram of Fig. 2.2 defines the navigation paths that the user can follow from the home page to the search results page and then to the details of the products. Also, the user is able to go back to the search results page from the detail of the product or go back to the home page.

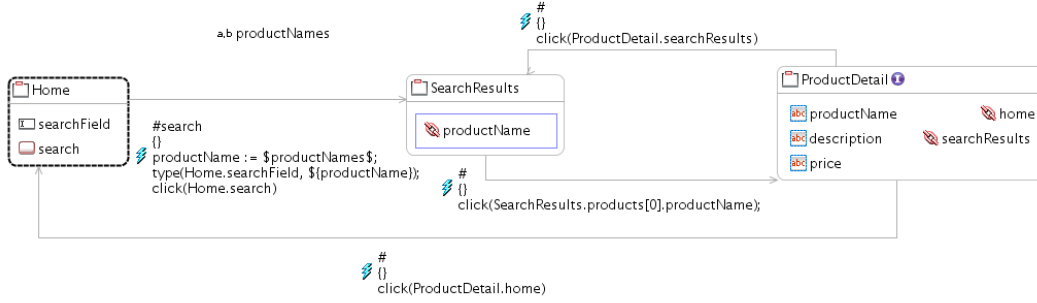


Fig. 2.2. Webspec diagram of the Search by name scenario

In a WebSpec diagram, an *interaction* represents a point where the user can interact with the application by using its interface objects (widgets). Formally, they represent the state of a Web page either when it is loaded after user's navigation or when it has changed as a consequence of a rich behaviour. Interactions have a name (unique per diagram) and may have widgets such as: labels, list boxes, buttons, radio buttons, check boxes and panels. Labels define the content (information) shown by an interaction. There are two types of widgets that allow defining widgets composition: ListPanel and Panel. A ListPanel represents a repetition of the elements that it contains and the Panel defines a simple placeholder that can contain any simple or composed widget. Interactions are graphically represented with a rounded rectangle (Fig. 2.3) which contains the interaction's name and widgets. A WebSpec diagram must have at least one starting interaction represented with dashed lines. To specify which properties must be satisfied by the application we use invariants (Boolean expressions) on the diagrams' interactions. Every interaction (either implicitly or explicitly) defines an invariant that specifies which properties must be satisfied in the set of scenarios specified by the diagram (in case that we do not define one explicitly, it is implicitly assumed that the invariant is true).

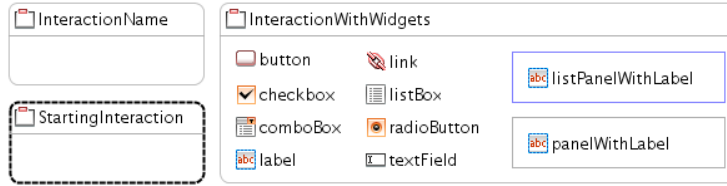


Fig. 2.3. WebSpec's interaction

In WebSpec, a navigation is graphically represented (Fig. 2.4) with grey arrows while its name, precondition and triggering actions are displayed as labels over them. In particular, its name appears with a prefix of the character “#”, the precondition between {} and the actions in the following lines. We must remark that the idea behind the transitions' actions (either navigations or rich behaviours) is that the execution of them produces the transition between interactions and not in the other way. A transition should be understood like: “if the precondition holds and the user executes the sequence of actions then the application should transit to the target interaction”.

A navigation from one interaction to another can be activated if its precondition holds, by executing the sequence of triggering actions such as: clicking a button, adding some text in a text field, etc. As well as invariants, preconditions can reference variables declared previously in the diagram. Actions are written according to the following syntax: `var := expr | actionName(arg1,... argn)` (a complete BNF [12] grammar can be found in Appendix A).

On the other hand, the application may change its UI state as a consequence of some actions performed by the user (e.g. on some interface widgets). For example, when the mouse

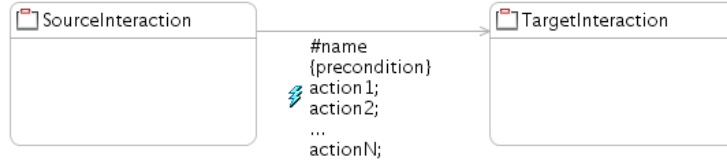


Fig. 2.4. WebSpec's navigation

is “on” a widget, some additional information might pop-up, or while entering text in a field, the text might be auto-completed. These “local” changes are common in the so-called rich Internet applications [13] and it is nowadays usual that customers pose requirements of this type, either explicitly (“I want an auto-complete feature in this field”), or implicitly (“I want that information appears as in Amazon.com”). These “rich” behaviours are being increasingly used not only in Web 2.0 applications but also in traditional, e.g. e-commerce, ones.

In a Web application, a rich behaviour is perceived by a local change in the UI of the Web application and it does not add a new element in the browsing history. To specify a rich behaviour in Webspec, we use a red dashed arrow (Fig 2.5) though it has the same properties that a navigation has (name, precondition and actions).

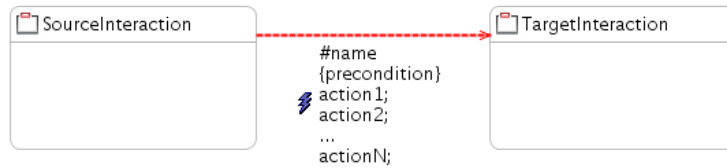


Fig. 2.5. WebSpec's rich behaviour

Improving requirement understanding using simulation

With the aim of improving the requirement elicitation phase, WebSpec diagrams allow the simulation of the application under development. Simulation is important to bridge the gap between the understanding of customers and analysts about requirements, thus helping to get real feedback from them. Usually, requirement artefacts [23] require some level of knowledge from customers to be fully understood, causing understanding problems during elicitation that are handled lately when the application is under active development.

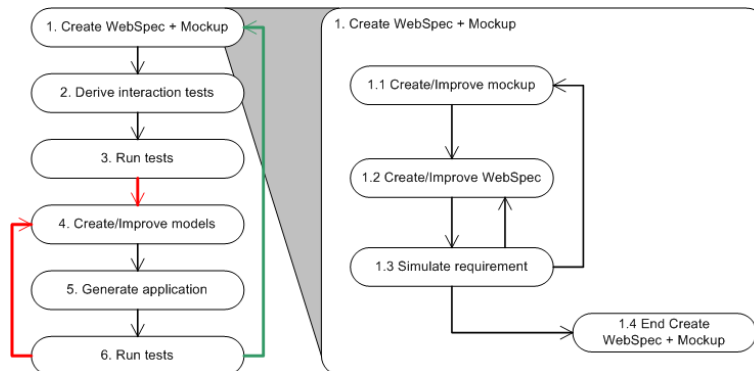


Fig. 2.6. WebSpec's simulation in the context of WebTDD

In WebTDD, simulation can be used during requirement gathering while we create the mockups and the diagrams. In Fig. 2.6 we show a detailed view of the “Create Mockups and WebSpec” activity; we first start creating some mockups which help to give a context of work to customers. Then we create the WebSpec diagrams according to the requirements from the customers and to double check the expected behaviour of the application we simulate some of their interaction paths. When we have agreed with the customer about the requirement, the “Create Mockups and WebSpec” activity ends.

To support the simulation of the application, WebSpec allows associating interactions with mockups and WebSpec widgets with their concrete UI elements in the mockup. Using this association, we can switch between the specifications in WebSpec with an exemplar UI that will help to understand the requirements. Mockups can be created with tools such as Balsamiq [2], Axure [1] or plain HTML. For example in Fig 2.7, we show a mockup of the product details page created with Balsamiq. The mockup shows the information that must be presented on that page: the product name, its description, price and the links to the home and search results. Fig. 2.8 shows a simple association between the mockup of Fig. 2.7 with its corresponding interaction and widgets.

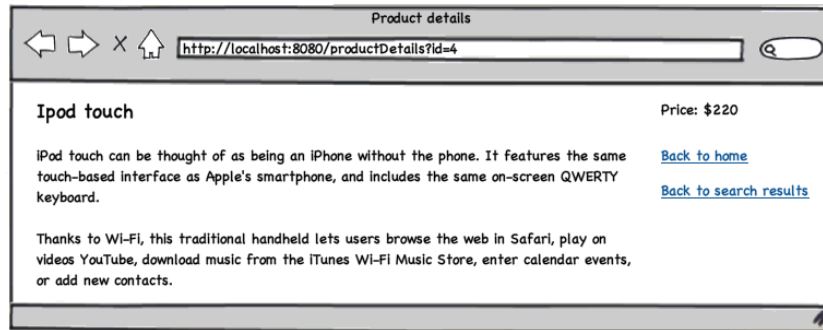


Fig. 2.7. Balsamiq mockup of Product details page

Our simulation basically opens a Web browser with the developed mockups and show descriptions and performs actions that show how a hypothetical user would interact with the application. It is rigorous, because differently from the simulation provided by tools such as Balsamiq [2], we not only show transitions between the pages but also execute real actions and provide descriptions of what would be the real output of the application, directly over mockups. These descriptions are generated automatically from the WebSpec diagrams, and they are easy to understand because they are written in natural language. In this way, from every WebSpec diagram, a set of simulations is automatically generated which can be used at any time by customers to understand the meaning of the diagram and suggest changes or improvements to the analyst.

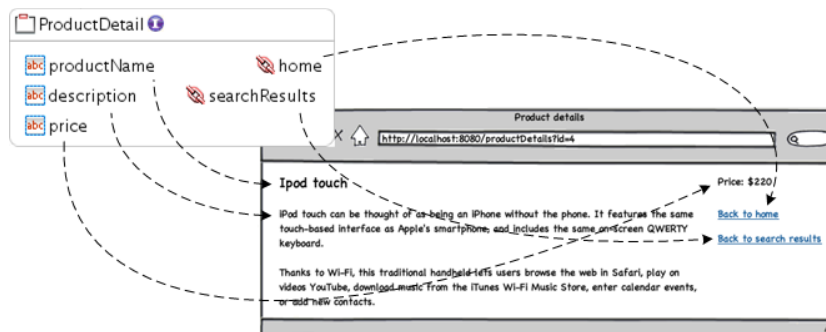


Fig. 2.8. Association between interaction and mockup

Validating requirement implementation with automatic test derivation

New requirements must be validated to guarantee their correct implementation while previous ones still work as intended. However, it is hard to perform this task efficiently, therefore keeping the requirements updated is extremely important.

A well known way of validating requirements consists in running automated tests (that express the requirements) over the application. If one of these tests fails, then a requirement is not satisfied by the application. In particular, interaction tests play an important role in industrial settings as they execute a set of actions in the same way a user would do it on a real Web browser, thus their use is continuously growing [21]. In Fig. 2.9 we show in more detail the activities performed during a WebTDD cycle; first we need to select the set of WebSpec diagrams that express the new requirement (Step 2.1) and automatically derive a set of interaction tests (Step 2.2). Afterwards, we run those tests by selecting the new test suite (Step 3.1) that has been derived and run them using an automated framework (e.g. JUnit) (Step 3.2).

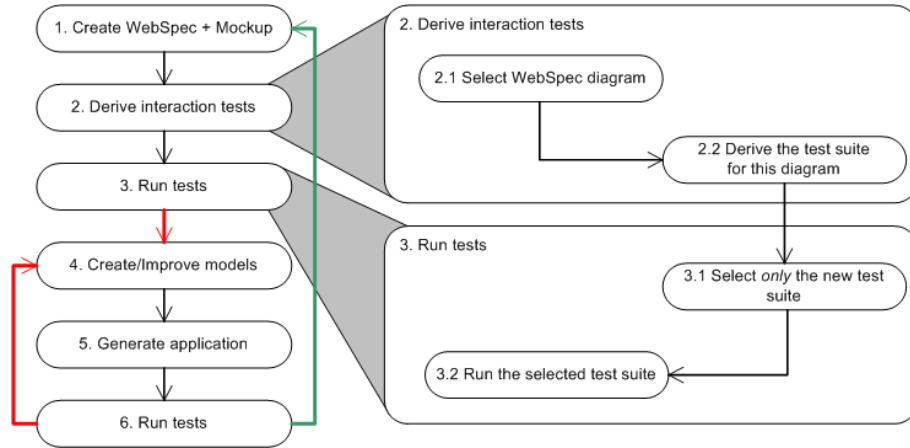


Fig. 2.9. WebSpec's test derivation in the context of WebTDD

The process of transforming WebSpec diagrams into test suites is automated and can be formally described in an algorithm that is applied over the diagrams (Chapter 4 and Appendix A). The algorithm follows these steps:

1. Create the test suite.
2. Compute all the possible paths of the diagram.
3. For each path:
 - a) Create a test class.
 - b) Open the URL of the initial interaction.
 - c) Add all the steps in the path from the initial interaction to the tail of the path including assertions for the invariants.

This transformation is technology agnostic and can be later refined into a technology dependent one (e.g. Selenium tests).

Semiautomatic application evolution using WebSpec change management

Capturing requirements changes is an important feature to predict their impact in the application. Though some mature requirement artefacts [17] provide extensions to support change management, in the Web engineering field this issue has been often ignored (see Chapter 4 and Appendix A for details).

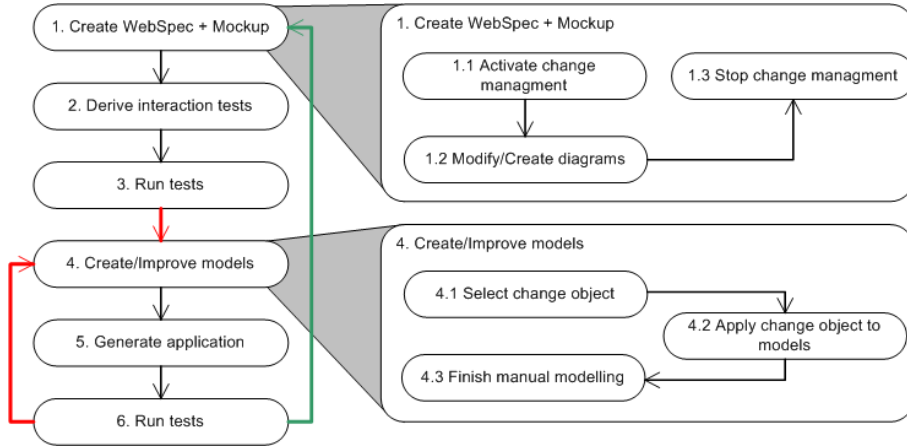


Fig. 2.10. WebSpec’s code derivation in the context of WebTDD

In WebSpec, changes are recorded into change objects that group a set of changes. Change objects are created even in the initial stage (when a diagram is being created). WebSpec diagrams can suffer different coarse grained changes, such as the addition or deletion of an interaction or transition element. These elements can be modified too, by the addition or deletion of widgets to an interaction, changes in invariants, etc. As for transitions, we can add or delete preconditions, change their source, target, or the actions that triggers them. When the user modifies the diagram, a change object is created and the sequence of changes is recorded as instances in a metamodel (Chapter 4 and Appendix A). In Fig 2.10 we present these activities in the context of WebTDD; when we are creating or modifying diagrams, we activate WebSpec’s change management to record these changes. Later, when we start with the modelling activities, we apply these changes automatically to our models to “upgrade” them. As WebSpec does not support all possible “upgrades” (specially those related with how the application has been modelled) we continue the modelling phase manually.

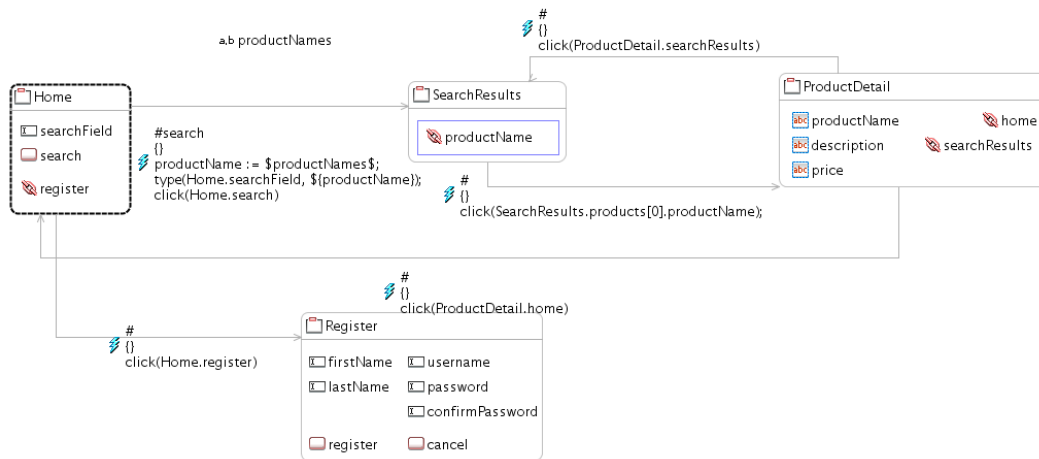


Fig. 2.11. Extension of the search diagram with a Register interaction

As an example, let us suppose that we add a Register interaction with its widgets and a link to it from the Home interaction (Fig. 2.11). The change in the diagram generates a new change object which has the following elements: the new interaction (Register), a new navigation (Home → Register), a new link (register) in the Home interaction and set of widgets in the Register interaction.

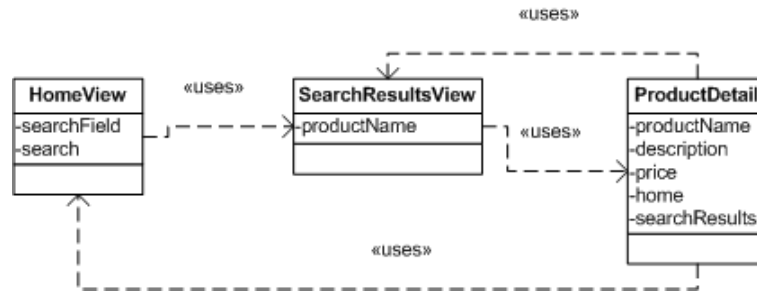


Fig. 2.12. Existing version of the UI model before applying the change object

Assuming we are modelling our UI with a class based model (Fig 2.12), we can upgrade it automatically to the one shown in Fig 2.13 using WebSpec change management (Chapter 4 and Appendix A).

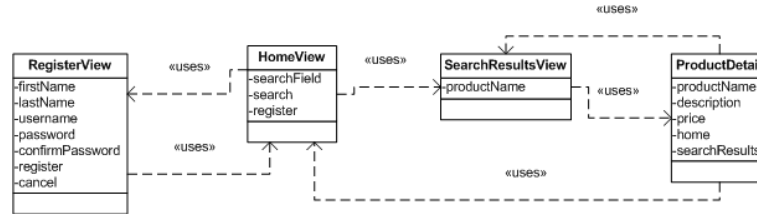


Fig. 2.13. Modified version of the UI model after applying the change object

2.3.3 Implementation

A WebSpec tool has been implemented as an Eclipse plugin using EMF [3] and GMF [4] technologies and it is currently available as an open source project¹.

The plugin supports the following features:

- Creation of WebSpec diagrams: a visual editor allows creating, modifying and updating diagrams. The properties of the elements can be modified by selecting each item and updating the property editors in the properties view.
- Association with HTML mockups: taking advantage of the Eclipse framework, HTML mockups are files inside the project. The editor allows selecting an interaction and associating it with the HTML file. Association between Webspec's widgets and HTML widgets is performed by editing the location property of Webspec's widget.
- Simulation of the application: Using the previous association, the plugin opens the mockups in the Web browser and show descriptions of what is the expected behaviour. This feature has been implemented by extending the Selenium [6] communication mechanism and using a JQuery plugin [5] for showing the descriptions.
- Selenium test derivation: As previously shown, each diagram is transformed into a test model. Then, the plugin allows the translation of the test model into a Selenium test.
- Change recording: Using the EMF observer pattern [15], we hook on all changes that are performed in the diagram and the plugin creates a change model. The user of the plugin can decide when should the plugin start recording changes and when not. When some changes are captured and the user stops recording, the change model is stored into a file for later use.

¹ See <http://code.google.com/p/webspec-language/> for further details

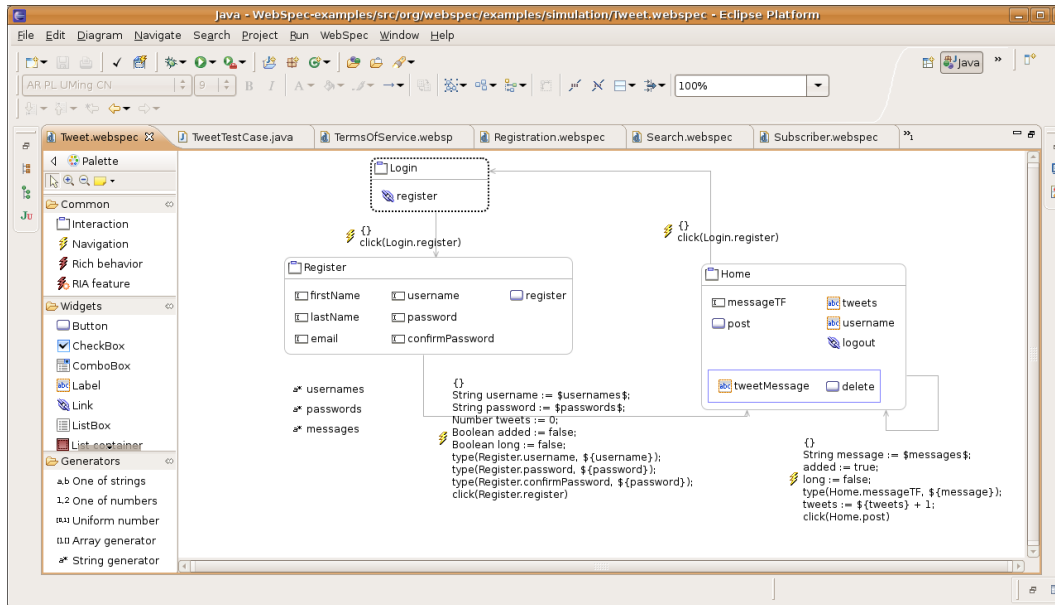


Fig. 2.14. WebSpec's Eclipse plugin

- Generation/Update of GWT and Seaside UI classes: Finally, using the previous stored change model, the UI model can be generated. Currently, the plugin allows the generation of GWT and Seaside classes and handles not only a first version of changes but also an incremental set of changes.

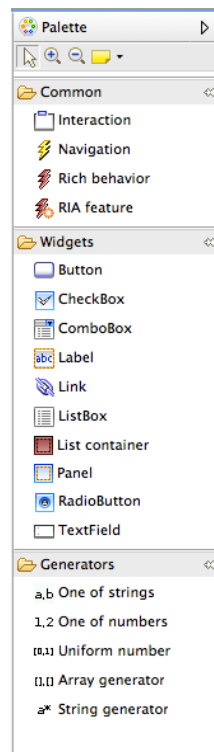


Fig. 2.15. WebSpec's palette

Fig. 2.14 shows a screenshot of the WebSpec’s Eclipse plugin. In Fig. 2.15 we can see in more detail WebSpec’s palette which allows the creating of each WebSpec element by simply drag and drop an element into the diagram. Then by selecting an element we can edit its properties using Eclipse’s properties view (Fig. 2.16). In the following subsections we provide more details regarding the implementation of the aforementioned features in the plugin.

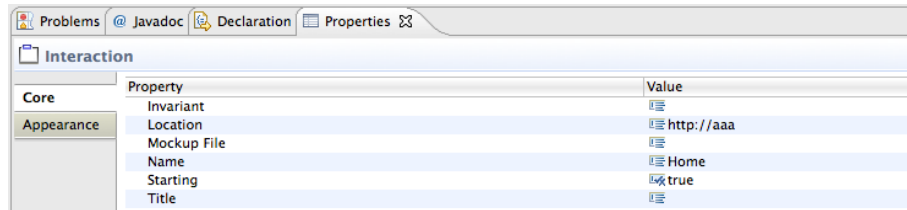


Fig. 2.16. WebSpec’s properties

Dealing with Simulation

The simulation feature comprises three elements: transformation between WebSpec and Simulation models, association with mockups and execution of the simulation. The transformation between WebSpec and the Simulation models has been implemented directly in Java as it was much simpler to deal with path computing algorithms than using QVT. To perform this transformation we simply open WebSpec’s menu (Fig. 2.17) and select Simulate.

Mockups association has been easily implemented by taking advantage of the Eclipse environment. We add a new property for interactions and widgets and a file dialog to let the user choose the right HTML mockup.

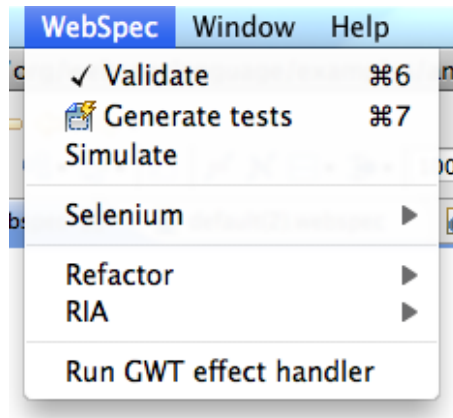


Fig. 2.17. WebSpec’s menu

The actual simulation aspect was more complex and required the extension to the Selenium framework. We used the existing communication mechanisms of Selenium to open the Web browser and execute actions. As shown in Fig. 2.18, we show descriptions over the mockups by using a JQuery plugin. To make it work, we had to extend the Selenium framework to load these libraries and actually show the descriptions when necessary. We must notice that the same mockup (which could be richer than the interaction since it has more widgets) could be used in multiple and different simulations. Our approach maintains the mockup as it is without removing any existing widgets because doing so will confuse the stakeholders about their presence or absence.

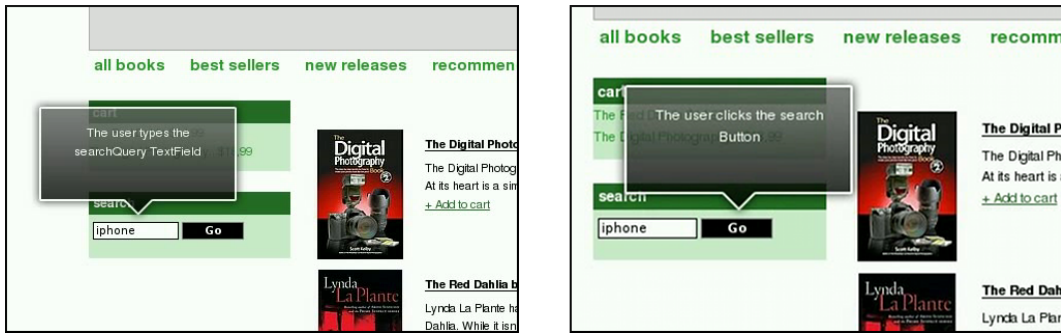


Fig. 2.18. WebSpec's simulation

Requirements validation

The support for requirements validation has been implemented in a two phase process: transformation from WebSpec to Test models, and test derivation to a specific automated test technology. The transformation between the models has been implemented by taking advantage of the existing simulation architecture (the transformation module), since both transformations use path computing algorithms.

In order to perform test derivation to a specific technology, we transformed the test models into a plain text representation of the test. The plugin currently supports the derivation to Selenium and we are working on the derivation to Webdriver [7]. As an example we show next the generated code for the Selenium framework for our example scenario:

```
(01) selenium.open("http://localhost:8080/index.html");
(02) selenium.type("id=searchField", "Ipod");
(03) selenium.click("id=search");
(04) selenium.waitForPageToLoad("30000");
(05) selenium.click("id=product0");
(06) selenium.waitForPageToLoad("30000");
(07) assertTrue(selenium.getText("id=productName").equals("Ipod"));
(08) selenium.click("id=home");
(09) selenium.waitForPageToLoad("30000");
(10) selenium.type("id=searchField", "book");
(11) selenium.click("id=search");
(12) selenium.waitForPageToLoad("30000");
(13) selenium.click("id=product0");
(14) selenium.waitForPageToLoad("30000");
(15) assertTrue(selenium.getText("id=productName").equals("book"));
(16) selenium.click("id=home");
```

Line 1 opens the application in the Web browser. Lines 02-04 search for Ipod product, lines 05-06 selects the first product and finally line 07 asserts that the selected product has the name Ipod. Lines 08-09 navigate to the Home page. Lines 10-12 search for book product, lines 13-14 select the first product and finally line 15 asserts that the selected product has the name book. Line 16 navigates to the Home page.

As an example, Selenium tests can be run in the Selenium IDE, Fig 2.19 shows a failing test exposing a requirement that has not been implemented by the application yet.

Requirement changes

When a diagram is modified, we record its changes and store them in change files. A change file is a serialization version of the change model in XML format. To capture the changes we

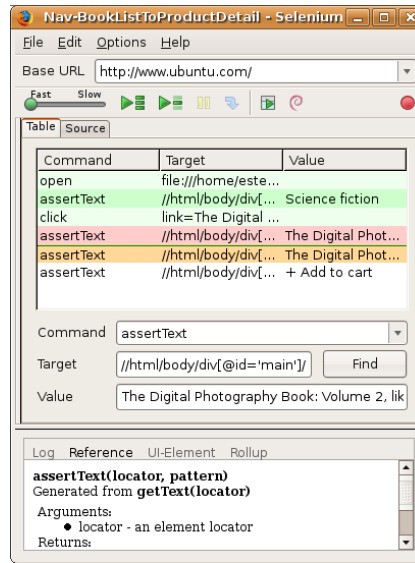


Fig. 2.19. Failing test

use the observer pattern and incrementally build the change model; afterwards we serialize it into an XML file.

Changes are read and used to upgrade the application models by effect handlers (a component that is able to map changes in the WebSpec level to technology ones). The plugin supports the generation of classes and methods compatible with Seaside and GWT, and we are actively working to provide a derivation to WebRatio design models [8].

As an example of the use of effect handlers, we next show how to use the change objects of our exemplar upgrade (Add a register functionality) to generate classes and methods in GWT. For the sake of conciseness we show the new RegisterView class created by the GWT effect handler.

Basically, lines 09-15 define the instance variables representing the widgets, and lines 21-29 initialize the objects with the proper GWT classes. Also, notice that RegisterView extends VerticalPanel (a GWT base class for implementing UIs).

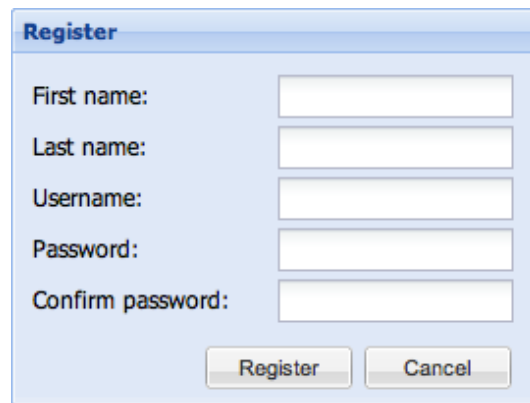
```
(01) package org.webspeclanguage.re;
(02)
(03) import com.google.gwt.user.client.ui.VerticalPanel;
(04) import com.google.gwt.user.client.ui.TextBox;
(05) import com.google.gwt.user.client.ui.Button;
(06)
(07) public class RegisterView extends VerticalPanel {
(08)
(09)     private TextBox firstName;
(10)     private TextBox lastName;
(11)     private TextBox username;
(12)     private TextBox password;
(13)     private TextBox confirmPassword;
(14)     private Button register;
(15)     private Button cancel;
(16)
(17)     public RegisterView() {
(18)         this.initializeComponent();
(19)     }
(20)
```

```

(21) public void initializeComponent() {
(22)     this.firstName = new TextBox();
(23)     this.lastName = new TextBox();
(24)     this.username = new TextBox();
(25)     this.password = new TextBox();
(26)     this.confirmPassword = new TextBox();
(27)     this.register = new Button();
(28)     this.cancel = new Button();
(29) }
(30) }

```

Fig. 2.20 shows a visual representation of the RegisterView class with some styling applied to improve the look and feel of the UI.



The image shows a window titled "Register" with a light blue header. Inside the window, there are five labels on the left: "First name:", "Last name:", "Username:", "Password:", and "Confirm password:". To the right of each label is a white text input box. At the bottom of the window, there are two buttons: "Register" and "Cancel".

Fig. 2.20. A visual representation of the RegisterView class

2.4 Conclusions

Web application development is a complex and time consuming process that involves different stakeholders with different knowledge and roles. In addition, it is common for these teams to face the challenge of evolving web applications in short periods of time to meet the new market requirements. Primarily, because upgrading the application according to the new requirements is a hard task if we want to avoid the problem of breaking existing functionality.

In the literature, the solution the facto has been MDWE approaches that use models to develop the application. However, these approaches are more heavy than agile ones and feedback from customers is obtained too late. On the other hand, agile approaches are code centric and requires a lot of manual effort for several tasks including web application testing.

To deal with these problems, we have presented in this PhD thesis a hybrid approach called WebTDD that mixes the advantages of MDWE approaches with agile ones. This approach has been the trigger to develop the main element of this thesis; a DSL for specifying Web requirements called WebSpec. We have shown how we can specify Web requirements using the language and at the same time simulate the application under development. Simulation is supported when it is used with Mockups and helps to improve the understanding of the requirement by the different stakeholders. As aforementioned, testing is crucial in this context and we have shown how a complete test suite is derived from each WebSpec diagram allowing to validate whether the requirement has been correctly implemented or not. Finally, taking advantage of the change management support that WebSpec provides, we have shown how we can upgrade the application under development in a semi automatic way.

It is worth to mention that WebTDD is the first hybrid approach to show that the combination between agile and model based approaches is feasible in the Web field. Also, WebSpec is the first DSL for specifying Web requirements that allows the aforementioned features and is independent of the development process. In this thesis we have used WebTDD because it is a perfect match for WebSpec's features.

References

1. Axure - wireframes, prototypes, specifications. available at: <http://www.axure.com/>.
2. Balsamiq. available at: <http://www.balsamiq.com/products/mockups>.
3. Eclipse emf. available at: <http://www.eclipse.org/modeling/emf/>.
4. Eclipse gmf. available at: <http://www.eclipse.org/modeling/gmp/>.
5. jquery: The write less, do more, javascript library. available at: <http://jquery.com/>.
6. Selenium web application testing framework. <http://seleniumhq.org/>.
7. Webdriver. available at: <http://webdriver.googlecode.com>.
8. The webratio tool suite. available at: <http://www.webratio.com>.
9. S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
10. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
11. S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks and Isdn Systems*, 33:137–157, 2000.
12. N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, Sept. 1956.
13. J. Duhl. Rich internet applications. a white paper sponsored by macromedia and intel, idc report, 2003.
14. M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
16. J. Gómez and C. Cachero. *OO-H Method: extending UML to model web interfaces*, pages 144–173. IGI Publishing, Hershey, PA, USA, 2003.
17. I. Jacobson. *Object-Oriented Software Engineering: a Use Case driven Approach*. Addison-Wesley, Wokingham, England, 1995.
18. N. Koch, A. Knapp, G. Zhang, and H. Baumeister. *UML-BASED WEB ENGINEERING - An approach based on standards*, chapter 7, pages 157–191. Springer, 2008.
19. D. Lowe. Web system requirements: an overview. *Requir. Eng.*, 8(2):102–113, 2003.
20. R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
21. E. M. Maximilien and L. Williams. Assessing test-driven development at ibm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society.
22. A. McDonald and R. Welland. Web engineering in practice, 2001.
23. D. L. Moody. The éphysicisé; of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Software Eng.*, 35(6):756–779, 2009.
24. O. Pastor, S. M. Abrahão, and J. Fons. An object-oriented approach to automate web applications development. In *Proceedings of the Second International Conference on Electronic Commerce and Web Technologies*, EC-Web 2001, pages 16–28, London, UK, 2001. Springer-Verlag.
25. E. Robles Luna, I. Garrigós, J. Grigera, and M. Winckler. Capture and evolution of web requirements using webspec. In *Proceedings of the 10th international conference on Web engineering*, ICWE'10, pages 173–188, Berlin, Heidelberg, 2010. Springer-Verlag.
26. E. Robles Luna, J. Grigera, and G. Rossi. Bridging test and model-driven approaches in web engineering. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Web Engineering*, volume 5648 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02818-210.
27. G. Rossi and D. Schwabe. Modeling and implementing web applications with oohdm. In G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, editors, *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pages 109–155. Springer London, 2008. 10.1007/978-1-84628-923-16.
28. M. Wimmer, A. Schauerhuber, and H. Kargl. On the integration of web modeling languages: Preliminary results and future challenges.

PhD Thesis as a Collection of Papers

A context for WebSpec: The WebTDD approach

The content of this chapter corresponds with the following papers:

Robles Luna E., Grigera J., Rossi G. *Bridging Test and Model Driven Approaches in Web Engineering. Proceedings of 9th International Conference on Web Engineering (ICWE 2009). 2009. San Sebastian, Spain. Acceptance rate: 24%. Core C.*

Robles Luna E., Panach J.I., Grigera J., Rossi G., Pastor O. *Incorporating Usability Requirements in a Test/Model-Driven Web Engineering Approach. Journal of Web Engineering (JWE). 2010. Impact factor: 0.531. JCR.*

This chapter describes the WebTDD approach: a test driven model based approach for web application development in which tests play a fundamental role driving the development process. Failing tests indicate the part of the required functionality that has not been implemented (similar to test driven development [10]). However, different from test driven approaches in which the main development artefact is the code, in our methodology we use a model based development in which models abstract but **not** drive the development.

The content of this chapter corresponds with the framework where we will applied WebSpec, the main contribution of this PhD thesis. As a reference we show in the figure below the structure that corresponds to this chapter.

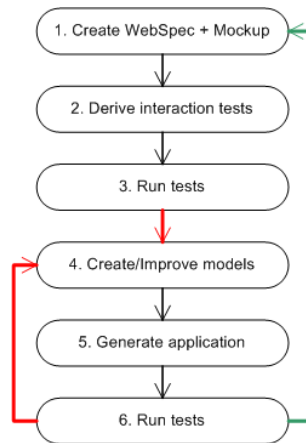


Fig. 3.1. The WebTDD approach

The content of this chapter is a paper published in *the International Conference of Web Engineering Support Systems (ICWE)* and another one in the *Journal of Web Engineering (JWE)*. ICWE aims at promoting research and scientific excellence on Web Engineering and at bringing together scientists and practitioners interested in technologies, methodologies, tools, and techniques used to develop and maintain Web-based applications. On the other hand, the JWE aims to provide a forum for advancing the scientific state of knowledge in all areas of Web Engineering. JWE articles address significant issues and problems, and potential solutions.

Bridging Test and Model-Driven Approaches in Web Engineering

Esteban Robles Luna^{1,2}, Julián Grigera¹, and Gustavo Rossi^{1,2}

¹ LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
{esteban.robles,julian.grigera,gustavo}@lifia.info.unlp.edu.ar

² Also at CONICET

Abstract. In the last years there has been a growing interest in agile methods and their integration into the so called “unified” approaches. In the field of Web Engineering, agile approaches such as test-driven development are appealing because of the very nature of Web applications, while model-driven approaches provide a less error-prone code derivation; however the integration of both approaches is not easy. In this paper, we present a method-independent approach to combine the agile, iterative and incremental style of test-driven development with the more formal, transformation-based model-driven Web engineering approaches. We focus not only in the development process but also in the evolution of the application, and show how tests can be transformed together with model refactoring. As a proof of concept we show an illustrative example using WebRatio, the WebML design tool.

1 Introduction

Agile methods [7, 16] are particularly appealing for Web applications, given their short development and life-cycle times, the need of small multidisciplinary development teams, fast evolution, etc. In these methods applications are built incrementally, usually with intense feedback of different stakeholders to validate running prototypes.

Unfortunately most solid Model-Driven Web Engineering (MDWE) approaches, even claiming to favor incremental and iterative development, use a more formal¹ and waterfall style of development. Web engineering methods like UWE [14], WebML [6], OOWS [18], OO-H [9] or OOHDM [22] define a set of abstract models such as content (called also data or application), navigation and presentation model, which allow the generation of running applications by automatic (error free) model transformations. This approach is attractive because it raises the abstraction level of the construction process, allowing developers to focus on conceptual models instead of code. The growing availability of techniques and tools in the universe of model-driven development (e.g. transformation tools) adds synergy to the approach.

¹ While Agile approaches might be also “formal” (see [7]), more popular ones tend to encourage a handcrafted style.

Many agile methods seem to follow a different direction. For example Test-Driven Development (TDD) uses small cycles to add behavior to the application [3]. The cycle starts with a set of requirements expressed with use cases [11] or user stories [13] that describe the application's expected behavior informally. The developer abstracts concepts and behavior, and writes a set of meaningful test cases which will fail on their first run, prior to the implementation. Then, he writes the necessary code to make the tests pass and run them again, until the whole test suite passes. The process is iterative and continues by adding new requirements, creating new tests and running them to check that they fail, then writing code to make them pass, and so on. In these cycles the developer might have to refactor [8] the code when necessary.

This strategy gives a good starting point for the development process, because developers specify the programs expected behavior first, making assertions about the return values right before the development itself begins. The process follows the idea of "Test first, by intention" [13], which is based on two key principles:

- Specify program's behavior (test first), and write code only when you have a test that doesn't work.
- Write your code without thinking about *how* to do a thing, instead think about *what* you have to do (intention).

Moreover, when using a static typed language like Java, the tests code may not even compile, as the involved classes and methods still don't exist. Thus, writing the tests first, guides us to create the classes and methods of the domain model. TDD allows better communication among different stakeholders, as short cycles favor the permanent evaluation of requirements and their realization in incremental prototypes. TDD is also claimed to reduce the number of problems found on the implementation stage [21] and therefore its use is growing fast in industrial settings [15].

In the Web Engineering area, efforts to integrate agile and model-driven development styles are just beginning [2], and most methods lack clear heuristics of how to improve the development life-cycle with the incorporation of these new ideas.

In this paper we present a novel, method-independent approach, to bridge the gap between TDD and MDWE approaches. The overall process has the same structure as TDD, but instead of writing code, we generate it from the well-known content, navigational and presentation models using a MDWE tool. We also create automated tests (that can be run without manual interaction) and deal with Web refactoring interventions [17]. These navigational and presentation tests allow us to manage evolution in a TDD fashion. Also, like in traditional TDD, we specify the application's behavior prior to its development in terms of tests, and use them to specify the application models, as they express (and validate) the expected functionality. We also relax some of the assumptions in TDD (based on its inherent bottom-up approach), as they are not appropriated for highly interactive applications. We illustrate our approach showing how to use these ideas in the context of the WebML methodology, using the WebRatio [24] tool.

The main contributions of the paper are the following:

- We present a novel TDD-like process to improve Model-Driven Web Engineering.
- We propose the use of black box interaction tests as essential elements for validating the application's navigational and interface behavior.

- We present an approach for dealing with navigation and interface test evolution during the refactoring process.

It should be noticed that our focus is in the development process and not in the tests themselves. Rather, we see tests as tools for driving the web application's construction and evolution.

The structure of the paper is the follows: In Section 2 we review some related work; In Section 3 we present our approach, and using a case study we explain how we map requirements into test models, and how the cycle proceeds after generating the application. We end the technical description of our approach by discussing, in Section 4 and 5, refactoring issues, both in the application and in the test models. Finally, we conclude and present some further work we are pursuing.

2 Related Work and Discussion

The advantages of using agile approaches in Web application development processes have been early pointed out in [16]. The authors not only argue in favor of agile approaches, but also propose a specific one (AWE) that, being independent of the underlying Web engineering method, could in theory be used with any of them. However, AWE is “just” a process; it does not indicate how software artifacts are obtained or how the process is supposed to be integrated in a model-driven development style.

Most Web Engineering methods such as WebML, UWE, OOHDM, OOWS or OO-H, have already claimed to use incremental and iterative styles, though support for specific agile approaches has not been reported yet in the literature.

In the broader field of software engineering, agile approaches have flourished, though most of them are presented as being centered in coding, much more than in the modeling and design activities. An interesting and controversial point of view in this debate can be found in [19], in which the author proposes to use an extreme “non-programming” approach, by only using models as development artifacts. In this arena, Test-Driven Development has been presented as one of the realizations of Extreme Programming [13], where tests are developed previously to code. In a recent paper [12] however, the authors clearly indicate that TDD is also appropriated as a design technique, and show examples in which TDD is used beyond “extreme” approaches.

The interest of using TDD in interactive applications is relatively new, given that the artifacts elicited from tests are usually “far” from the interface realm, and also because unit testing [4], which focuses on individual classes, is unsuitable for complex GUIs. In [1], the authors present a technique for organizing both the code and the development activities to produce fully tested GUI applications from customer stories. Similarly, [20] proposes to use TDD as an approach to develop Web applications, focusing on the development of the different parts of the MVC triad, again emphasizing coding more than modeling.

Also, in relation to our approach, as TDD makes a heavy use of requirements models, it is important to say that most Web engineering approaches have either automatic ways or explicit heuristics to derive content and navigation models from requirements documents; particularly, in OOWS [18], the conceptual model can be generated from requirements using model-to-model transformations; earlier in [5], the

authors have presented an attractive way to map use cases into navigation models in the context of OO-H and UWE, giving much more relevance to the requirement documents. The concept of Navigation Semantic Unit in [5] has inspired our idea of Navigation Unit Testing (see Section 3).

In a different direction, though still related with our ideas, [10] show how to systematically generate test cases from requirements, particularly from use cases. These proposals however deal with tests as usual in non-agile processes, therefore running them against a “final” application, instead of profiting from them during the whole development process.

3 An Overview of Our Approach

In the TDD approach, new functionality is specified in terms of automated tests derived from individual requirements, and then the code to make them pass is written. A further step involves refactoring this code by removing duplication, for example. Obviously TDD does not deny the need to perform a thorough testing process of the final application; the tests in TDD are a perfect start to assess how the application fulfills the client’s requirements beyond its correctness.

Our approach follows the same structure, but given the nature of Web applications instead of focusing on unit testing, we emphasize the use of navigation and interaction level tests, which we first run against user interface (UI) mockups using a black box approach. We then replace the coding by a modeling step, generating the code using a MDWE tool. We also add an intermediate step to adapt the tests, in order to trim the differences between the mockups and the generated application prototype.

Even though we face application generation using MDWE tools, this stage of our process differs slightly from the conventional model-driven approach, as we work at a very fine granularity level: in the extreme case, we build models for one requirement at a time, generating tested and running prototypes incrementally, leading each requirement through a lightweight version of a full MDWE step. In this way, we come closer to the TDD short-cycle style, while still profiting from the advantages of working with models.

Briefly explained, our approach mixes TDD and MDWE techniques to make Web development more agile. We first gather user requirements with use cases [11], User Interaction Diagrams (UIDs) [22] and presentation mockups [25]. Then, we choose a use case and derive an interaction test against the related presentation mockup, with which we specify the navigation and UI interaction prior to the development. We next get a running prototype of the application by creating models and generating code in a short MDWE cycle, and check its correctness using the test. Should these tests fail, we would go back to tweak the models, regenerate the application and run them back again, repeating the process until they pass. As in TDD, the complete method is repeated with all use cases, until a full-featured prototype is reached. Fig. 1 shows a simplified view of our approach, confronting it with the “traditional” TDD.

While the application evolves, tests will also help to check that functionality is preserved after applying navigation and presentation refactorings (i.e. usability improvements that don’t alter the application behavior [17]).

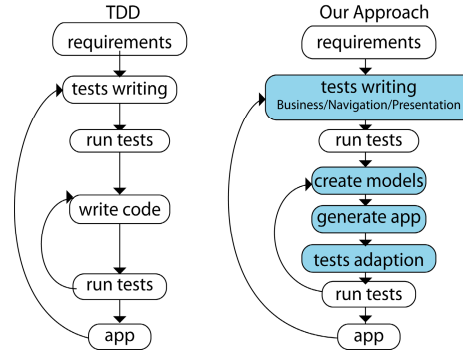


Fig. 1. TDD life cycle comparison

In the following subsections we illustrate the approach with the development of TDDStore, a simplified online bookstore, similar to Barnes&Noble. As we use WebML and WebRatio, which support data-intensive applications, we focus mainly on navigation and UI tests, also contemplating some business operations.

3.1 Capturing Requirements with Mockups and UIDs

Similarly to a MDWE approach, we begin gathering and modeling the set of requirements. Particularly, we propose employing use cases, UIDs and mockups. With these artifacts, the analyst can easily specify UI, navigation and business requirements that the application must satisfy. For each use case, we specify the corresponding UID that serves as a partial, high-level navigation model, and provides abstract information about interface features. As an example of an interaction diagram, we show in Fig. 2 the UID corresponding to the case when the user is presented with a list of books, indicated with “...” in state 1, containing some information about each book (“title, author...”), and selects a book from the list (transition marked with 1) to see the full book details (state 2).

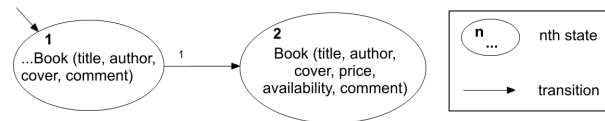


Fig. 2. UID for simple navigation

Using UI mockups, we agree with the client on broad aspects of the application look and feel, prior to the development. This is a very convenient way for interacting with stakeholders and gathering quick feedback from them. There are two additional reasons to use UI mockups: we will perform UI and navigational tests against them, and they will become the application’s final look and feel.

In Fig. 3.a we show an initial and simplified mockup of our application’s main page, where all books are listed. Fig. 3.b shows a mockup for the book details page. In the

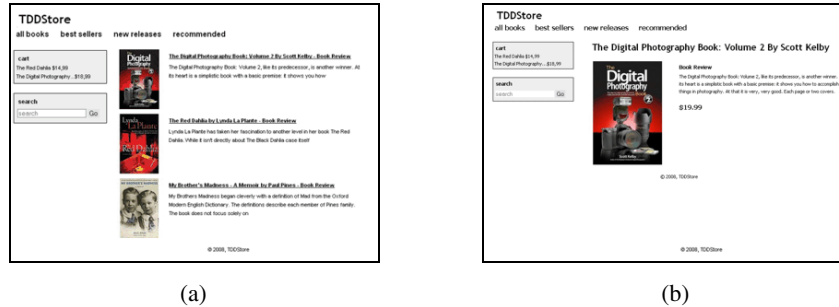


Fig. 3. a) Books list mockup; b) Book details mockup

next sub-section we show how to specify a test against this mockup to verify the UID in Fig. 2. To make the example realistic, we also included some other features in the mockup, though they will be tested in further iterations, when being involved in a use case and UID.

3.2 Writing Tests

Mockups and UIDs help to understand the expected behavior of the application. UIDs refine use cases to show how the user interacts with the application, and mockups complement UIDs to give a sample of the application look and feel. However, these useful tools fall short to provide by themselves an artifact capable of being run to validate the application's expected behavior. By incorporating interaction tests, we provide a better way to validate the application.

Following the process we create a test for the mentioned use case, using as a basis the UID in Fig. 2 and the mockup in Fig. 3. For the sake of clarity and concreteness instead of an abstract test specification, we tie our description to a standard test tool like Selenium [23], to specify the interactions between the user and the application (other similar tools can be used for this task). These tests rely on the DOM structure of the tested document, so they are agnostic of the process by which the application has been generated, as well as the applied styles. The following test validates that the UI shows the book list and the navigation between the book list and the book's detail:

```
public class BookListTestCase extends SeleneseTestCase {
    public void testBookListToProdDetailNav() throws Exception {
(1)  sel.open("file:///dev/bookstore/Mockups/books-list.html");
(2)  assertEquals("all books", sel.getText("//div[@id='tb']/p[1]"));
(3)  sel.click("link=The Digital Photography Book");
(4)  sel.waitForPageToLoad("30000");
(5)  sel.assertLocation("/bookDetail*");
(6)  assertEquals("The Di...", sel.getText("//div[@id='prod']/h2"));
(7)  assertEquals("The ...", sel.getText("//div[@id='p-d']/p[1]"));
(8)  assertEquals("+ Add to...", sel.getText("//div[@id='p-d']/a"));
    }
}
```

The test begins by opening the page (the mockup file) (1) and asserting that a specific element has some content (2); in this way we can assert that we are in the book list page. Then we specify to click on a specific link (3) and wait until the page

is loaded (4) and validate our location (5) thus validating our navigation. Then, we assert that several html elements contain the specific text (6-8) which validates that the UI has changed. When we try to run the test using the Selenium runner it fails because we have not yet developed the running application. This scenario is the same as in TDD where the test is expected to fail after it has been written.

These tests are similar to traditional unit tests but performed on small “navigation units” arising from a single use case, so we call them navigation unit tests.

This kind of tests simulate user interactions (click on a link, fill a text box, etc.) and add assertions about the elements of the page. Navigation unit tests are independent of the MDWE tool used because they run using a web browser. We found this type of tests suitable for testing most of the business, navigation and UI logic as perceived by the user. However, in complex Web applications there are many scenarios in which unit and integration tests [4] (the usual TDD type of tests) should be used. One example is the integration between Web applications using Web services. Another one are application’s behaviors performed “in the shadows” (e.g. support for the shipping process in an e-store). In both cases, interaction tests are not useful because the user might not be interacting with the application. We don’t include these examples as illustrations as they are not novel in TDD. For these tests our approach remains unchanged: specify a test (e.g. a unit test), check that it fails, specify the corresponding models (e.g. using WebML units, UWE classes, etc.), generate the application, etc.

At this point, we can start using our design artifacts (mockups and UIDs) to derive the application, navigation and presentation models.

3.3 Deriving Design Models

Once requirements have been (at least partially) gathered, and the tests specified for a particular use case, the next step is to generate a running application. As mentioned before, here is where we differ from a pure TDD approach, as we chose to use a MDWE tool, instead of writing code. Throughout the development of our proofs of concept we have used the WebML’s MDD tool, WebRatio [24]. We will concentrate on the navigational (hypertext) model for several reasons; first, it is the distinctive model in Web applications; besides we want to emphasize the differences between typical TDD and TDD in Web applications and show how navigation unit tests work. Additionally, as said before, WebRatio’s (and WebML) content model is a data and not an object-oriented model, thus some of the typical issues in TDD (originally devised to work with classes and methods) do not apply exactly as they were conceived, as we discuss below.

A first data model is derived using the UIDs as a starting point, identifying the entities needed to satisfy the specified interactions, e.g. by using the heuristics described in [22]. As Web Ratio supports the specification of ER models at this stage of the development, the application behavior will be specified later, in the so-called logic model. Following with our example, we need to build an application capable of listing books, and exhibiting links to their corresponding details pages, so the book and author entities come out immediately from the UID in Fig. 2. Then, we map the navigation sequence in the UID to a WebML hypertext diagram, as shown in Fig. 4.

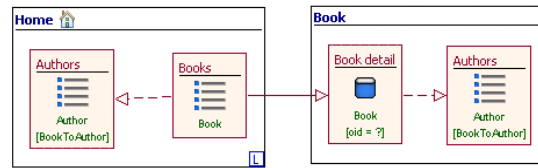


Fig. 4. WebML diagram for the UID

WebRatio is now ready to generate the application. Once we have a running prototype, we can adapt the tests (this process is detailed in section 3.4) and run them to check if the models (and therefore the application) conform with the requirements.

Finally, we need to adjust the application's presentation. WebML does not define a presentation model; instead presentation is considered like a document transformation from a WebML specification of a page into a specific language page like JSP or ASP.NET. In another methodology, the mockups and UIDs would be used to also specify the presentation model. Since we already had developed mockups for our current UID, this part of the process is straightforward: we only need to slice up the mockup, and input it as an XHTML template into WebRatio. We can run the tests again to ensure no interaction is corrupted while the template is being modified.

3.4 Test Adaptation

After building the models, we need to make sure the implementation generated from them is valid according to the requirements specification. In particular, we want to confirm that business, navigation and UI behavior are correct with respect to the tests defined in section 3.2. However, if we try to run the tests as they are written, they will fail because they still reference mockups files, and although the layout may be the same, the location in terms of an XPath expression [26] may have changed.

On one hand, the generation may have renamed the URLs of each page. For instance, if we chose to transform templates into JSP pages, URLs change their names to end with ".jsp". We can prevent this scenario by defining the name of the mockups upfront, according to the technology. Another problem may arise if we use components that generate HTML code in a different way than what we had expected. We face this problem, for example, when we display a collection of objects using WebRatio's Table component. This could be also prevented by using a customized template, in which we manually iterate over the collection of objects.

Although both scenarios could be prevented, we should consider the case in which they are not. In that situation we must adapt the test to the current implementation. Fortunately, the adaptation of tests is easy to perform manually, and its mechanics can be automated in a straightforward way. As an example, we show how to adapt the test of section 3.2 to be compliant to the current implementation.

```
public class BookListTestCase extends SeleneseTestCase {
    public void testBookListToProdDetailNav() throws Exception {
(1)  sel.open("http://127.0.0.1:8180/TDDStore/pagel.do");
(2)  assertEquals("all...", sel.getText("//div[@id='pagelFB']/p[1]"));
(3)  sel.click("link=The Digital Photography Book");
    }
```



```

(4) sel.waitForPageToLoad("30000");
(5) sel.assertLocation("/page2*");
(6) assertEquals("The ...", sel.getText("//div[@id='p2FB']/h2"));
(7) assertEquals("The D...", sel.getText("//div[@id='p2FB']/p[1]"));
(8) assertEquals("+ Add to...", sel.getText("//div[@id='p2FB']/a"));
    }
}

```

In the above test we first changed the URL to start the test by just finding the right URL and changing it (1, 5). Then, as the layout of the list of products has changed due to the derivation process of WebRatio, the XPath expressions we had used are no longer valid as WebRatio has included a different DOM structure. This can be changed for example by accessing the url with a tool such as the XPather plugin [27]. Just right click over the item, shown in XPather and then copy the XPath expression to the test (2, 6-8). Next we can re-run the test, and verify it succeeds.

3.5 Towards a New Iteration

Having our iteration complete (i.e. all tests run correctly), we are ready to add new functionality to the application. We will incorporate the possibility of adding a book to a shopping cart, so we go through the same steps of the first example:

1. Model the new requirements, with use cases and UIDs.
2. Create a new mockup if necessary, or extend a previous one.
3. Write a new navigation unit test for the added functionality and run it against the corresponding mockup.
4. Upgrade the model and generate the application, implementing the new functionality to make the tests pass.
5. Adapt the new test, as previously shown in section 3.4
6. Run the new test and check that the new functionality has been correctly added. If the test fails, then go back to step 3 until it passes.

In order to introduce the new add-to-cart functionality we need to illustrate the interaction with a new UID (1) that slightly extends the one in Fig. 2 with a new navigational transition with the product being added to the cart. We need to expand the book details mockup by adding an "add to cart" link (2). Then we write the test in the same way as we did previously on section 3.2.

```

public class BookListTestCase extends SeleneseTestCase {
    public void testAddBookToShoppingCart() throws Exception {
(1) sel.open("file:///dev/bookstore/Mockups/books-list.html");
(2) assertEquals("The D...", sel.getText("//div[@id='p-i']/h4/a"));
(3) sel.click("//div[@id='product-info']/a");
(4) sel.waitForPageToLoad("30000");
(5) assertEquals("The Dig...", sel.getText("//ul[@id='s-p']/li[1]"));
(6) sel.assertLocation("/cart*");
    }
}

```

The test above opens the book list (1) and asserts the name of the product. Then clicks on the “add to cart” link of the product (3) and waits for the page to load (4). It asserts that the selected book has been added to the cart by asserting that the book's title is present in the shopping cart page (5) and that navigation has succeeded (6).

As we show in Fig. 5, an extended WebML hypertext diagram including the AddToCart operation is derived from the new UID.

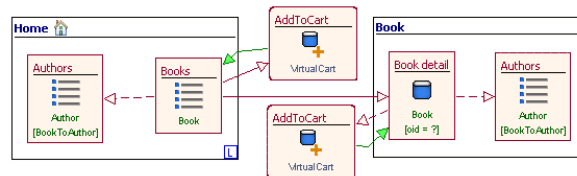


Fig. 5. Upgraded WebML diagram

We regenerate the application and run the whole test suite against the derived application. Notice that the test suite will be composed of the previously adapted test, and the new one after the corresponding adaptation.

4 Dealing with Application Evolution

Web applications tend to evolve constantly and in short periods of time; the evolution is driven mainly by two reasons:

- **New requirements:** Generally, new requirements arise because of clients or users' requests to enhance the application's functionality. For example, the book store's owner may want to categorize books, which would require defining new model elements (entities, page types, links, etc).
- **Web refactorings:** We might want to improve the application's usability, by either modifying the interface or the navigation facilities. This kind of model changes, usually driven by non-functional requirements (usability, accessibility, etc), have been called elsewhere Web model refactorings [17]. Web refactorings may eventually occur in a TDD cycle, for example if the developer notices an opportunity to improve the user experience.

Next, we analyze both cases and show how we deal with them during the test-driven development process.

4.1 New Requirements

After the application has been deployed (or even during its development), the client may want to add new functionality, such as organizing books in categories. New requirements have to be described using the artifacts we have previously mentioned (UIDs, mockups) and following the process we have summarized in Section 3.5:

1. Add the label that shows the category name of the book, to the mockup of books list and books' details.
2. Add the assertions to the adapted tests of the books list and books' details pages, with the XPath expression obtained from the mockups.
3. Run the tests and ensure they fail.

4. Enhance the domain, navigation and the UI models (entities, units and templates in WebRatio) to show the category.
5. Generate the application.
6. Run the tests (adapt them if necessary). If they fail go back to step 4.

After finishing this cycle, we will have a new requirement added to the application and a new test that validates the UI of the book list and book detail pages. Obviously, we might want to navigate through categories but the process remains similar just by adding some new use cases and UIDs before 2 and building the corresponding tests.

4.2 Web Refactorings

Web refactorings seek to improve application's usability with small model changes. A catalog of such refactorings has been presented in [17]. In order to illustrate the process we selected a fairly simple one, *Turn Information into Link*, which consists in converting a text string into a link leading to a page with information about the object represented by the text. In our case, we will enhance the authors' names on the book details page and transform them into links, leading to a list of their books. Once again, we will follow the steps of our approach as follows:

1. Refactor the book details mockup to show a link where each author name appears, as shown in Fig. 6.

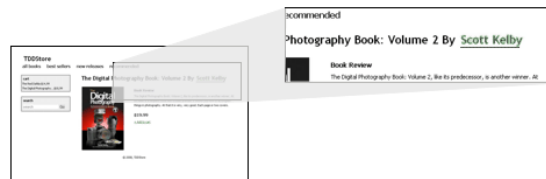


Fig. 6. Refactored book details mockup

2. Transform the UI test of the book detail page (3) by changing the XPath expression. Previously it was an h2 element, but now it is a link, so we have to change it to an a element. Also, add a test to validate the navigation from the book detail to the author page (8-13).

```
public class BookDetailTestCase extends SeleneseTestCase {
    public void testBookDetailUI() throws Exception {
(1) sel.open("http://127.0.0.1:8180/TDDStore/page2.do?oid=2");
(2) assertEquals("The ...", sel.getText("//div[@id='p2FB']/h2[1]"));
(3) assertEquals("Sc...", sel.getText("//div[@id='prod-d']/a"));
(4) assertEquals("Book R...", sel.getText("//div[@id='p2FB2']/h3"));
(5) assertEquals("The ...", sel.getText("//div[@id='p2FB2']/p[1]"));
(6) assertEquals("$19.99", sel.getText("//div[@id='p2FB2']/p[2]"));
(7) assertEquals("+ Add t...", sel.getText("//div[@id='p2FB2']/a"));
    }
    public void testBookDetailNavigationToAuthor() throws Exception {
(8) sel.open("file:///dev/bookstore/Mockups/books-detail.html ");
(9) assertEquals("Scott Kelby", sel.getText("//div[@id='p-d']/a"));
(10) sel.click("//div[@id='p-d']/a");
    }
}
```

```

(11) sel.waitForPageToLoad("30000");
(12) assertEquals("Books f...", sel.getText("//div[@id='p-1']/h2"));
(13) sel.assertLocation("/byAuthor*");
    }
}

```

3. Run the tests and ensure they fail.
4. Modify the corresponding WebML hypertext model and the corresponding presentation
5. Derive the application.
6. Run the tests (adapt them if necessary). If they fail go to step 4.

At the end of this cycle we have a complete refactoring applied over the application and tests transformed and added to the test suite. We next show how we can automate this kind of tests transformations.

5 Towards Automated Test Evolution

During the development cycle, “old” tests should always succeed (except that some already processed requirement has changed dramatically). However, Web refactorings pose a new challenge for the developer: even not being originated by new requirements, they can make navigation tests fail, as they might (slightly) change the navigational and/or interface structure of the application. In other words, and as shown in 4.2, tests must be adapted to be useful after a refactoring, i.e. to correctly assess if it was safely performed. Fortunately, refactorings can be catalogued, because, as well as design patterns, they record and convey good design practices. Therefore, it is feasible to automate the process of test transformation. This refactoring-driven transformation of tests must be performed after the mockup and UIs have been modified to show the new expected behavior. To transform a test we need to follow these steps:

1. Select the test transformation associated with the refactoring of the catalogue to be applied.
2. Configure the test transformation with UID's, mockups, location of tests and specific parameters of the transformation (e.g. a specific element's location).
3. Apply the test transformation.

There are many strategies to transform tests; we next explain one of them, as it comprises defining a model for tests, which can be useful for other further tasks, such as linking tests' components to design model elements, for example to improve traceability. To achieve automatic tests transformation, we first need to abstract the concepts involved in a Web test. A Web test is a sequence of interactions and assertions that aim to validate the application's behavior. An interaction allows the user to interact with the application. For example: click a link, click a button, type a text on an input field, check a checkbox, etc. Assertions allow ensuring that a predicate is valid in the current context. There are many possible assertions over a Web page such as `assertTitle`, `assertTextPresent`, etc. A Web test could be then abstracted using the simplified model shown in Fig. 7.

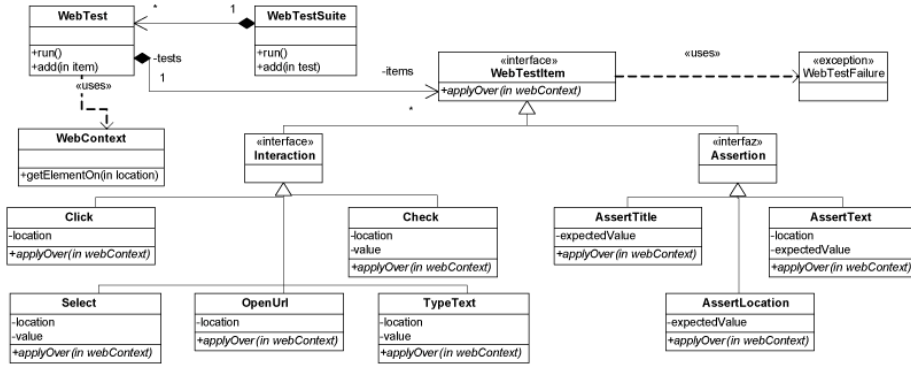


Fig. 7. Web Test Model

Individual tests can be abstracted, from their source code to an instance of the model, in a straightforward way by using a parser. When tests are mapped onto a set of objects, they can be easily manipulated. For instance, adding a title assertion to a test is as simple as creating a new instance of the AssertTitle class and adding it to the WebTest instance. Web test transformations are then designed and coded with objects, and thus the algorithm that performs the transformation can be coded and encapsulated in a class. Once the test transformation has been applied, we translate objects back into the test text using a pretty printing algorithm. We omit here the explanation of the parsing and pretty printing phases, as they are outside the scope of the paper. As an example we show the algorithm of the Turn Information Into link [17] test transformation that can be summarized in the following steps:

1. Request the location of the test.
2. Request the location of the text.
3. Change the location of the AssertText instance of the text. If no assertion is pointed by the user, create a new instance of the AssertText class.
4. Create a new WebTest instance. Create an OpenUrl instance (pointing to the mockup) and clone the AssertText instance of 3. Add both instances to the WebTest.
5. Create a Click and Wait instances pointing to the location of the new link and add it to the WebTest instance.
6. Request the expected location and a text that identifies the new location.
7. Create an AssertText and AssertLocation instances with the corresponding requested values.

The result of applying the algorithm looks similar to the result shown in section 4.2, but instead of testBookDetailNavigationToAuthor, the new test is called testNavigationTextToLink1. Using this approach we can automate the process of Web test transformation based on the catalogue of refactorings we want to apply.

6 Concluding Remarks and Further Work

We have presented a novel approach to integrate test-driven development into model-driven web engineering methods. Our approach can be used with any of the existing methods, though to illustrate its feasibility we have used WebML and WebRatio as a proof of concept. We have briefly explained the main steps of our approach and showed some advanced aspects, such as tests transformations during the Web refactoring stage. We have also shown that most activities related to tests evolution can (and indeed should) be automated. To our knowledge, our proposal is the first to bridge the gap between model-driven approaches and test-driven development, and particularly in the Web engineering field. We retain the agile style of TDD that focuses on short cycles, each one aimed at implementing a single requirement, to validate the generated prototype. However, we work at a higher level of abstraction (i.e. with models) leaving code generation to the support tool.

While TDD is usually, due to its strong relationship with coding, a handcrafted and therefore error-prone activity, integration with model-driven approaches opens an interesting space for improvement. We are now working on several directions: first we are making field experiences to measure the impact of the integration on development costs and quality aspects. While both TDD and model-driven development improve software construction, we believe that our approach tends to synergize the benefits more than just summing them up. From a more technical point of view we are working in the integration of tools for TDD in different MDWE tools. These tools include: Selenium and XPath for developing test cases, and Selenium RC to make a one click away the generation and running of the whole test suite (currently done manually). We are also planning to use an object-oriented approach (like UWE), together with its associated tool to research deeper in the relationships between typical unit testing in TDD (focused on object behaviors) and our navigation unit testing, which focuses more on navigation and user interactions. Automatic generation of tests from UIDs by using transformations or strategies like the one described in [10], and improving traceability between tests and models are also important items in our research agenda.

References

1. Alles, M., Crosby, D., Erickson, C., Harleton, B., Marsiglia, M., Pattison, G., Stienstra, C.: Presenter First: Organizing Complex GUI Applications for Test-Driven Development. In: AGILE 2006, pp. 276–288 (2006)
2. Ambler, S.W.: The object primer: agile modeling-driven development with UML 2.0. Cambridge University Press, Cambridge (2004)
3. Beck, K.: Test Driven Development: By Example. Addison-Wesley Signature Series (2002)
4. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1999)
5. Cachero, C., Koch, N.: Navigation Analysis and Navigation Design in OO-H and UWE. Technical Report. Universidad de Alicante, Spain (April 2002), <http://www.dlsi.ua.es/~ccachero/papers/ooH-uwe.pdf>
6. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. Computer Networks and ISDN Systems 33(1-6), 137–157 (2000)

7. Eleftherakis, G., Cowling, A.: An Agile Formal Development Methodology. In: SEEFM 2003 Proceedings 36 (1 de 12) (2003)
8. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Reading (1999)
9. Gómez, J., Cachero, C.: OO-H Method: extending UML to model web interfaces. In: van Bommel, P. (ed.) Information Modeling For internet Applications, pp. 144–173. IGI Publishing, Hershey (2003)
10. Gutiérrez, J.J., Escalona, M.J., Mejías, M., Torres, J.: An approach to generate test cases from use cases. In: Proceedings of the 6th international Conference on Web Engineering. ICWE 2006, Palo Alto, California, USA, July 11 - 14, vol. 263, pp. 113–114. ACM, New York (2006)
11. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. ACM Press/Addison-Wesley (1992)
12. Janzen, D., Saiedian, H.: Does Test-Driven Development Really Improve Software Design Quality? IEEE Software 25(2), 77–84 (2008)
13. Jeffries, R.E., Anderson, A., Hendrickson, C.: Extreme Programming Installed. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2000)
14. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-Based Web Engineering, An Approach Based On Standards. In: Web Engineering, Modelling and Implementing Web Applications, pp. 157–191. Springer, Heidelberg (2008)
15. Maximilien, E.M., Williams, L.: Assessing test-driven development at IBM. In: Proceedings of the 25th international Conference on Software Engineering, Portland, Oregon, May 03 - 10, pp. 564–569. IEEE Computer Society, Los Alamitos (2003)
16. McDonald, A., Welland, R.: Agile Web Engineering (AWE) Process: Multidisciplinary Stakeholders and Team Communication. In: Web Engineering, pp. 253–312. Springer, US (2002)
17. Olsina, L., Garrido, A., Rossi, G., Distant, D., Canfora, G.: Web Application evaluation and refactoring: A Quality-Oriented improvement approach. Journal of Web Engineering 7(4), 258–280 (2008)
18. Pastor, O., Abrahão, S., Fons, J.: An Object-Oriented Approach to Automate Web Applications Development. In: Bauknecht, K., Madria, S.K., Pernul, G. (eds.) EC-Web 2001. LNCS, vol. 2115, pp. 16–28. Springer, Heidelberg (2001)
19. Pastor, O.: From Extreme Programming to Extreme Non-programming: Is It the Right Time for Model Transformation Technologies? In: Bressan, S., Küng, J., Wagner, R. (eds.) DEXA 2006. LNCS, vol. 4080, pp. 64–72. Springer, Heidelberg (2006)
20. Pipka, J.U.: Test-Driven Web Application Development in Java. In: Objects, Components, Architectures, Services, and Applications for a Networked World, vol. 1, pp. 378–393. Springer, US (2003)
21. Rasmussen, J.: Introducing XP into Greenfield Projects: lessons learned. IEEE Softw. 20(3), 21–28 (2003)
22. Rossi, G., Schwabe, D.: Modeling and Implementing Web Applications using OOHDM. In: Web Engineering, Modelling and Implementing Web Applications, pp. 109–155. Springer, Heidelberg (2008)
23. Selenium web application testing system, <http://seleniumhq.org/>
24. The WebRatio Tool Suite, <http://www.Webratio.com>
25. VanderVoord, M., Williams, G.: Feature-Driven Design Using TDD and Mocks. In: Embedded Systems Conference Boston (October 2008)
26. XML Path Language (XPath), <http://www.w3.org/TR/xpath>
27. XPather - XPath Generator and Editor, <https://addons.mozilla.org/en-US/firefox/addon/1192>

INCORPORATING USABILITY REQUIREMENTS IN A TEST/MODEL-DRIVEN WEB ENGINEERING APPROACH

ESTEBAN ROBLES LUNA ^{2,3}, JOSÉ IGNACIO PANACH¹, JULIÁN GRIGERA²,

GUSTAVO ROSSI ^{2,3}, OSCAR PASTOR¹

¹*Centro de Investigación en Métodos de Producción de Software*

Universidad Politécnica de Valencia

Camino de Vera s/n, 46022 Valencia, Spain

{jpanach, opastor}@pros.upv.es

²*LIFIA, Facultad de Informática, UNLP, La Plata, Argentina*

{esteban.robles, julian.grigera, gustavo}@lifia.info.unlp.edu.ar

³Also at Conicet

Received November 13, 2009

Revised May 26, 2010

The success of Web applications is constrained by two key features: fast evolution and usability. Current Web engineering approaches follow a "unified" development style which tends to be unsuitable for applications that need to evolve fast. Moreover, according to the quality standard ISO 9126-1, usability is a key factor to obtain quality systems. In this paper, we show how to address usability requirements in a test-driven and model-based Web engineering approach. More specifically, we focus on usability requirements with functional implications, which do not only concern the visual appearance, but also the architecture design. Usability requirements are contemplated from the very beginning of each cycle, by creating a set of meaningful tests that drive the development of the application and ensure that no functionality related to usability is altered unintentionally through development cycles. Dealing with those usability requirements in the very early steps of the software development process avoids future hard changes in the system architecture to support them. The approach is illustrated with an example in the context of the OOWS suite.

Key words: Test-Driven Development, Usability, Conceptual Models,
Model-Driven Development

Communicated by: M. Gaedke & A. Ginige

1 Introduction

Developing quality Web applications quickly and error free is one of the most challenging problems in the web engineering field. This kind of software always stresses development teams because requirements tend to change fast (the "permanent beta" syndrome) [25]. At the same time, customers require extremely usable applications more than in other kind of software. As a consequence, it is reasonable to use a development process with short cycles and intense participation of stakeholders.

Agile development methodologies, such as Test-Driven Development (TDD) [3, 16], are a perfect match to this development style.

TDD uses short cycles to incrementally add behaviour to the application. Each cycle starts by gathering requirements in the form of Use Cases [19] or User Stories [21] that describe the expected behaviour of the application informally. Next, the developer abstracts concepts and behaviour, and concretizes them in a set of meaningful test cases. Those tests are intended to fail on their first run, showing that the application does not meet the requirements yet. In order to fix this problem, the developer writes the necessary code to pass the tests and runs them again until the whole test suite passes. The process is iterative and continues by adding new requirements. In these cycles, the developer can refactor [14] the code when it is necessary. Studies have shown that TDD reduces the number of problems in the implementation stage [33] and therefore its use is growing fast in industrial settings [26].

However, one of the problems of TDD is its extremely informal nature in which most design decisions remain undocumented. While TDD favours agility, it also hinders evolution in the middle and long term.

One attractive alternative to standard “code-based” TDD is to use a model-driven software development (MDSD) approach, which allows focusing on higher level design models and deriving code automatically from them, at the same time minimizing errors and making the development process faster [15]. However, MDSD Web engineering approaches [23, 6, 13, 17, 35] tend to use a “unified” [20] rather than an agile approach. To make matters worse, both agile and MDSD approaches lack a “natural” way to specify requirements dealing with usability, which as mentioned before is a key aspect in the Web engineering field.

While agile and MDSD-based approaches appear to be confronted frequently, our view is that their positive properties should be put to work together in order to provide more efficient and effective software production methods. This is why in this paper we present a novel development approach which aims to solve the problems discussed above: it is agile, can interplay seamlessly with model-driven approaches and supports specification and testing of usability requirements. Our approach combines the recent work of the authors in two different areas: test-driven development of Web applications [34], and specification and modelling of usability requirements [30].

On the one hand, we propose injecting a test-driven development style into a model-driven development methodology, developing the initial ideas presented in [34]. In this way, we maintain the agility of test-driven development while working at a higher level of abstraction by using models. One contribution of the presented work is to show that Agile and MDSD can be combined to make them become stronger together than separately. The approach begins building interaction and navigation tests derived from presentation mockups (i.e. stub HTML pages) and User Interaction Diagrams (UIDs) [35]; these tests are later run against the application generated by the model-driven development tool to check whether they pass or fail. On the other hand, we derive usability tests, i.e. those that capture the properties needed to build a usable system. These tests are used in the same way as “conventional” functional tests in the TDD cycle, thus serving as a way to check how development proceeds by formalizing one of the typical customers concerns. Another value of the presented work is to demonstrate that usability requirements can be properly dealt with in such an advanced software production process. The whole approach is complemented with a set of tools to simplify the stakeholders’ tasks.

To develop this idea, our work focuses on functional usability requirements, called in the literature Functional Usability Features (FUF) [22]. Historically, usability has been considered as a non-functional requirement [7]. However, many authors have discovered several usability properties strongly related to functionality [2, 12]. FUFs are usability requirements related to functionality and therefore related to system architecture. Each FUF is divided into different subtypes called usability mechanisms. Several authors propose including these mechanisms from the very early steps of the software development process in order to avoid changes in the architecture once they have been designed [2, 12]. Following the proposal to deal with usability in the early steps, we have used a set of guidelines defined by Juristo [22] to capture functional usability requirements for each usability mechanism. These templates contain a set of questions that the analyst must use to capture usability

requirements by means of interviews with the client. From these templates, we have extracted the usability properties that must be taken into account when the analyst develops the system.

In a brief summary, the contributions of the paper are the following:

- We show how to introduce usability requirements in an agile model-driven Web engineering approach.
- We illustrate the detection of some relevant properties to build usable systems. These properties have been extracted from templates to capture usability requirements defined in the literature.
- We show how to translate those properties into a set of meaningful tests that drive the development process.

The structure of the paper is the following: In Section 2 we discuss some related work in this area, covering Web development approaches and specification of usability requirements. In Section 3 we present the background of our approach. In Section 4 we show the approach in a step by step way, showing with small examples how we intermix test and model-driven development with a strong bias to usability checking. In Section 5 we present a lab case. Finally, in Section 6 we conclude and present some further work we are pursuing.

2 Related Work

Our proposal brings model-driven and agile approaches together, in an effort to improve Web development. Classical model-driven Web engineering methods like WebML [6], UWE [23], OOHD [35], OOWS [13] or OOH [17] usually favour a cascade style development. We superimpose a specific agile approach, Test-Driven Development, where tests are developed before the code (in this case the model) in order to guide the system development. In this sense, some authors like Bryc [4] have proposed generating these tests automatically, while in other works tests are constructed manually [26]. Both techniques are valid for our proposal.

We state, like Bass [2] and Folmer [12], that usability must be included from the very early steps in the software development process (TDD in our proposal). In other words, usability must be considered from the requirements capture step. Several authors, like Juristo [22], have dealt with usability as a requirement. Juristo has defined a set of Functional Usability Features that are related to system architecture. The requirements of these features are captured by means of guidelines. These guidelines include questions that the analyst must ask to end-users in order to adapt the features to users' requirements. Lauesen [24] also includes usability in the requirements capture, discussing six different styles of usability specification and showing how to combine them in a complex real-life case to meet the goals. The styles specify the usability properties more or less directly. The list of styles is: performance style; defect style; process style; subjective style; design style; guideline style. The best choice in practice is often a combination of the styles, so that some usability requirements use one style and others use a different one. Finally, it is important to mention the work of Cysneiros [8], who has defined a catalogue to guide the analyst through alternatives for achieving usability. The approach is based on the use of the *i** [42] framework, having usability modelled as a special type of goal. Cysneiros states that a catalogue can be built to guide the requirements capture. This notation provides a total view of requirements and the relationships among them, as well as the relationship between usability and functional requirements inclusively. The main disadvantage of this proposal is the *i** notation which is ambiguous, is far from natural language, and it may present contradictions [11]. The difference between our proposal and the aforementioned works is the context of use. We deal with usability requirements in a TDD process using a model-driven Web engineering approach, while the mentioned authors deal with usability requirements in a traditional software development process.

The concept of pattern is one of the most widely used concepts to include usability in the first steps of the software development process. Many authors have worked on the definition of usability patterns, for instance Tidwell [38]. The patterns described by Tidwell represent not only usability, but also interaction. The notation used to represent the patterns is graphical because Tidwell wants the user

to participate in the design of the architecture. Following the same trend as Tidwell, Perzel describes a set of patterns that are oriented to web environments [32]. Perzel distinguishes between patterns for web applications (users must introduce data) and patterns for web sites (users only navigate and visualize information).

One work that aims to bring usability patterns closer to the end user is carried out by Welie [40]. The patterns of Tidwell and Perzel differ from the patterns of Welie in that Welie distinguishes between the user perspective and the design perspective. The main reason for this sorting into groups is that, from the user perspective, it is important to state *how* and *why* the usability is improved; while from the design perspective, patterns only solve designer problems.

The patterns proposed by all these authors include a short description about the implications of including the patterns in the architecture. However, this description is too short. The patterns should have a guideline to explain in detail how to include the patterns in the system. In our proposal, that inclusion is hidden for the analyst, because it is performed by automatic transformation of the MDSD process.

Others authors like Nielsen [29], have been working recently on including usability in agile software development methods. Nielsen states that fast and cheap usability methods are the best way to increase user experience quality, because developers can use them frequently throughout the development process. This work is very close to our proposal, but it is not focused on a TDD approach. Again, the originality provided by our presented work is centered around its integration of TDD and MDSD, together with the incorporation of usability requirements in this approach.

Regarding automated testing within model-driven software development processes, it is important to mention the work of Dihn-Trong et al. [9] who apply validation techniques directly to UML models [39]. The authors create Variable Assignment Graphs (VAGs) to automatically generate test input, considering also the model's constraints. Nevertheless, generating VAGs requires the models to be already created, so it is not possible to guide the development through generated tests. Also, we state that users must participate in the test definition, but Dinh-Trong proposes testing the system by means of design models, where users cannot take part for ignorance.

In a recent work [34] we have illustrated a first attempt to apply our TDD-based methodology on a MDSD Web engineering approach, but usability was not considered. In the same way, Zhang [45] has presented an approach in which he applies Extreme Programming practices into a process, in a methodology called test-driven modelling (TMD). Tests are created in terms of message sends (represented as Message Sequence Charts) to a black box system, and then models are created to pass these tests. The overall approach is similar to the one here presented, but it does not consider navigation or presentation (hence, neither usability) early in the process.

Back on the agile track, Agile Model Driven Development (AMDD) [1] proposes a MDSD-like development process, but creating models that are “just barely good enough” to fulfil a small set of requirements. Our approach takes the same philosophy in that matter, but AMDD differs from it since it is not purely model-based, but it also has a latter coding stage in which TDD is applied. Other authors, such as Wiczorek [44], have proposed testing the system in the code generated from a Conceptual Model, as we propose. This author proposes black-box testing that uses structural and behavioural models described in UML to automatically generate test cases. After automatically generating part of the code from the Conceptual Model, developers are starting to create unit tests for the functions that they are going to implement. Changes derived from testing are applied directly to the code. This fact differs from our proposal, where changes are directly applied to the Conceptual Model and the code is automatically generated, making the software development process more efficient.

3 Bridging Usability requirements with TDD

We want to emphasize that our approach [34] puts together the advantages of both agile and model-driven approaches, and it is our strong belief that this is the path to be followed by modern software production approaches. Incorporating usability requirements in that domain is a concrete way to improve the quality of the associated method, as usability is a recognized quality software criteria. To achieve this goal, we deal with presentation mockups and requirements models early in the

requirements elicitation stage, integrating usability requirements in the incremental development process with a TDD style. By using a model-driven development approach we raise the level of abstraction; as a consequence, the quality of the generated software is significantly improved like in most MDWE approaches [15]. Instead of following a cascade style of development, we use an iterative and incremental style following short cycles constantly involving stakeholders. Usability requirements are taken into account from the earliest stages, and considered “first class” requirements for test generation; therefore they participate in the development cycle, just as regular requirements. We next describe our approach in detail.

3.1 The approach in a nutshell

The development cycle is divided into cycles or sprints (Figure 1). At the beginning of the sprint, the development team has only a set of short informal specifications (descriptions) of what they have to do. These specifications have been defined by means of interviews with the user. Developers start working by picking one of them at a time. A small cycle starts by capturing a more detailed analysis using informal requirement artefacts (Step 1). A variety of artefacts can be used depending on the type of requirement we are capturing:

- For requirements involving interactions, we use UIDs that serve as a partial specification of the application’s navigation. Mockups are used for User Interface (UI) aspects, and Use Cases (UC) or User Stories (US) for business or domain aspects.
- For usability requirements, we use a set of usability properties derived from usability requirements guidelines defined in the literature [22]. These usability properties are represented by: UIDs for navigational concerns and mockups for UI aspects. If functionality slightly changes, then UC/US must be used too.

The artefacts we use to capture requirements are described in natural language, lacking a clear/formal definition. Therefore, developers transform these requirements into tests, to get a more “formal” specification (Step 2). As in a TDD, tests are heavily used both to drive the development process and to check that existing functionality is not altered during the development process. This has probed [28] to reduce development time because unintentional errors are captured during the development cycle instead of leaving their discovery to the quality assurance (QA) or testing phase.

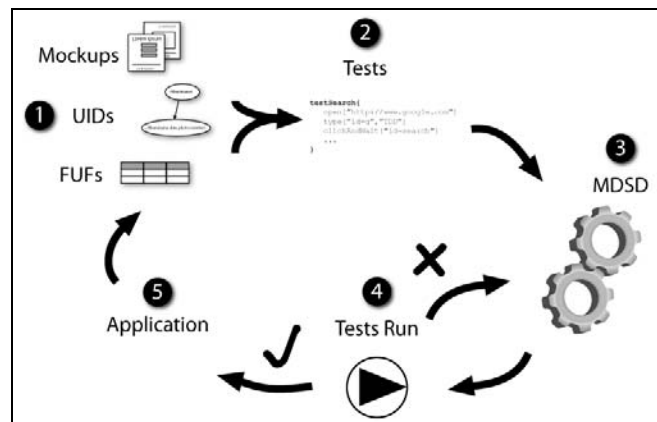


Figure 1 A Schema of our method

Before a requirement is implemented, corresponding tests must be run to check whether or not the application fulfils the requirement. The failing tests show which requirements are not yet supported by the system under development. If, at this stage, the application passes all the tests, then either they do not express the new requirement properly, or the new requirement is not new because the application already supported it. In the first case, we should give more detail to tests going back to step 2 and in the second one, we should dismiss the requirement and return to step 1.

Once the requirement has been specified in a test suite, the development phase can begin. By using a MDSD approach, the developer creates or extends the existing models generating an enhanced version of the application (Step 3). All the development effort is concentrated on building/extending the model. The code generation is performed automatically by means of transformation that takes as input the models.

In order to check that the requirement has been successfully implemented and no previous functionality is corrupted, the developer runs the whole suite of tests to check both things (Step 4). If one or more tests fail, he should go back to step 3, do some rework in the models, generate the code again, and retry step 4 until all tests pass.

Finally, we get a new application with one requirement added (Step 5). The cycle continues by picking a new requirement (Step 1) and following steps 2 to 5 until we run out of requirements for the sprint.

3.2 An overview of involved artefacts

Throughout the requirements elicitation activity, we combine different artefacts to achieve fluency in the communication between stakeholders, and accuracy in the specification for the development team. As we just stated in 3.1, UIDs, HTML mockups, FUFs and interaction tests help in both aspects. The first two are useful in terms of communication at early stages of requirements definition: UIDs provide a precise and somewhat intuitive way to specify navigation and interaction, while mockups reveal the presentation, making it concrete for customers. Usability needs are also detected at this early stage, following standardized guidelines by applying Functional Usability Features. Whenever possible, these requirements must be stated early in the process, since they might have an influence in the application's design (particularly in navigation and interface/interaction issues, but also in functionality). Finally, interaction tests come to play right before development, when a thorough specification that considers all possible ways of interaction stated in the UID diagrams is required. Additionally, they are specified against the same mockups obtained in the requirements gathering activity, to make sure the same interaction agrees with the stakeholders.

Along the following subsections, we explain each artefact with more detail, illustrating them with examples in the context of a simple, conventional e-commerce application that is useful to introduce the basic ideas. In section 5, we will introduce in more detail a concrete example related to a library management system.

3.2.1 Mockups

HTML mockups are simple, static web pages that act as sketches of the application. They are intended to be developed quickly to reflect the customer wishes in terms of presentation in a much more substantial way than requirements expressed in written language alone. Mockups show no difference from regular HTML pages, in fact their only characteristic feature is the way of building them and their use. However, they can eventually become useful in the final stages of the development, where the same mockups can become the definitive look and feel of the application.

In a simple E-commerce application, suppose that the customer explains that the checkout process must ask for the credit card information, and let the user revisit the list of products involved in the purchase. Figure 2 shows two mockups, Figure 2.a shows a product's detail page, from which the user can navigate to the checkout page shown in Figure 2.b.

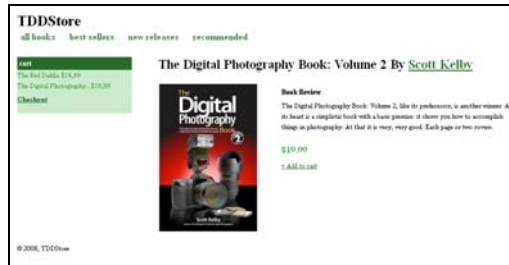


Figure 2a Product Detail Mockup

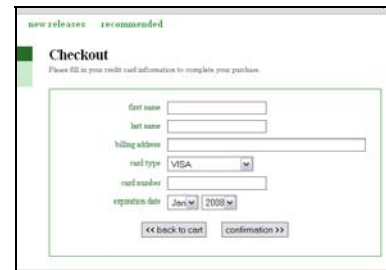


Figure 2b Checkout Mockup

3.2.2 User Interaction Diagrams

Similar to use cases, a UID describes the exchange of information between the user and the system and particularly the set of interactions occurring to complete a functional requirement. UIDs enrich use cases with a simple graphical notation to describe partial navigation paths. UIDs are simple state machines where each interaction step (i.e. each point in which the user is presented with some information and either indicates his choice or enters some value) is represented as an ellipse, and transitions between interaction points as arcs. For each use case, we specify the corresponding UID that serves as a partial, high-level navigation model, and provides abstract information about interface features. Following the example from 3.2.1, Figure 3 shows a simple UID expressing the operation of checkout, from the list of products through the confirmation.

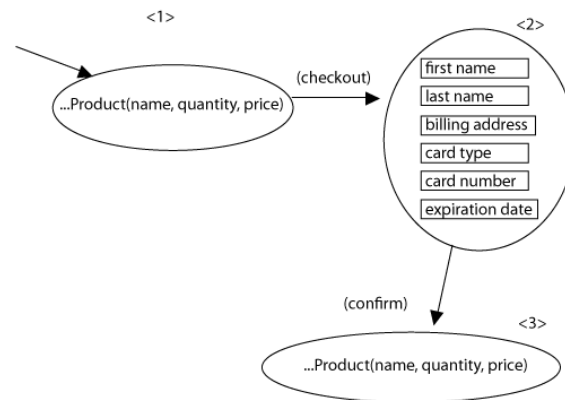


Figure 3 Checkout's UID

We have extended the UID notation to allow automatic generation of interaction tests as described below.

3.2.3 Interaction tests

An interaction test is a test that opens a Browser and executes a set of actions directly on it, in the same way a user would do it. Also it allows making assertions on HTML elements based on XPath [41] or HTML IDs. There are several tools that could be employed to write such a test: Selenium [36], Watir [43], TestNG [37]. The main advantage of this kind of tests is that they execute directly in the browser making them independent of the development method or tool used to generate the application.

Like in "conventional" TDD, we write the tests before the development begins. Using Selenium, we can speed up the code writing by profiting from the Selenium recorder to record the set

of actions we perform over the mockup. Later, we can translate the tests to Java and add the necessary assertions to ensure the application's expected behaviour.

These tests are also used after the requirement has been developed to check that the new requirement has been correctly implemented and previous functionality has not been altered.

3.2.4 FUF

Usability is a very wide concept. Human-Computer Interaction literature provides three types of recommendations to improve the usability of a software system [22].

1. Usability recommendations with impact on the user interface (UI). These recommendations refer to presentation issues with slight modifications of the UI design (e.g. buttons, pull-down menus, colours, fonts, layout).
2. Usability recommendations with impact on the development process. These can only be taken into account by modifying the whole development process. For example, those that intend to reduce the user cognitive load require involving the user in the software development.
3. Usability recommendations with impact on the architectural design. These involve building certain functionalities into the software to improve user-system interaction. These set of usability recommendations are referred to as Functional Usability Features (FUFs). FUFs are defined as recommendations to improve the system usability that have an impact on the architectural design. Examples of these FUFs are providing cancel, undo and feedback facilities.

We have focused our proposal on FUFs because a big amount of rework is needed to include these features in a software system unless they are considered from the first stages of the software development process [2, 12]. Therefore, the inclusion of FUFs must be done from the requirements capture step.

Different HCI authors [40, 38, 18] identify different varieties of these usability features. These subtypes are called usability mechanisms. In other words, each FUF has a main goal that can be specialized into more detailed goals called usability mechanisms.

3.3 OOWS: A Model-Driven Web Engineering Method

As said before, we favor the use of a MDSD style. Though the overall approach is independent of the specific MDSD methodology, we will illustrate the paper with OOWS. OOWS (Object-Oriented Web Solutions) [13] is a model-driven web engineering method that provides methodological support for web application development. OOWS is the extension of an object-oriented software production method called OO-Method [31], as Figure 4 illustrates. OOWS introduces the diagrams that are needed to capture web-based applications requirements, enriching the expressiveness of OO-Method. OO-Method is an Object Oriented (OO) software production method that provides model-based code generation capabilities and integrates formal specification techniques with conventional OO modelling notations. OO-Method is MDA compliant [27], so following the analogy with MDA, OO-Method provides a PIM (Platform-Independent Model) where the static and dynamic aspects of a system are captured by means of three complementary views, which are defined by the following models:

- **Structural Model** that defines the system structure and relationships between classes by means of a *Class Diagram*.
- **Dynamic Model** that describes the valid object-life sequences for each class of the system using *State Transition Diagrams*.
- **Functional Model** that captures the semantics of state changes to define service effects using a textual formal specification.

As Figure 4 shows, OOWS introduces two models in the development process:

- **Navigational Model:** This model describes the navigation allowed for each type of user by means of a Navigational Map. This map is depicted by means of a directed graph whose nodes represent navigational contexts and their arcs represent navigational links that define the valid navigational paths over the system. Navigational contexts are made up of a set of **Abstract Information Units (AIU)**, which represent the requirement of retrieving a chunk of related information. AIUs are made up of **navigational classes**, which represent views over the classes defined in the Structural Model. These views are represented graphically as UML classes that are stereotyped with the «view» keyword and that contain the set of attributes and operations that will be available to the user. Basically, an AIU represents -at a conceptual level- a web page of the corresponding Web Application.
- **Presentation Model:** The purpose of this model is to specify the visual properties of the information to be shown. To achieve this goal, a set of presentation patterns are proposed to be applied over conceptual primitives. Some properties that can be defined with this kind of patterns are information arrangement (register, tabular, master-detail, etc), order (ascendant/descendent) or pagination cardinality.

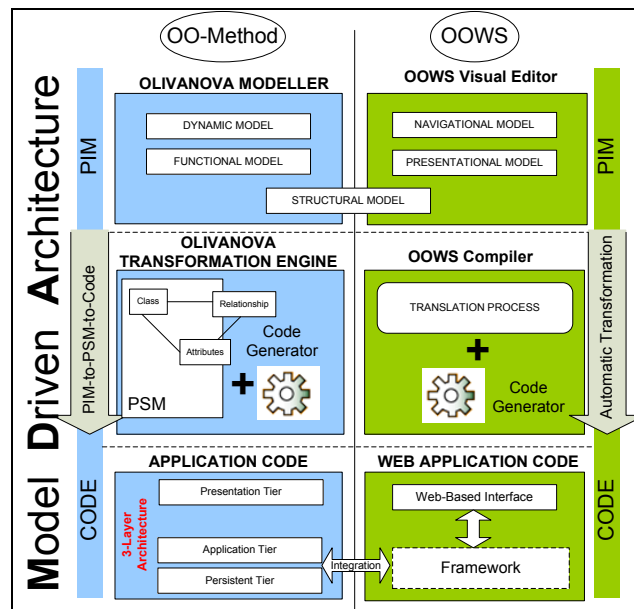


Figure 4 OO-Method and OOWS MDA Development Process

Both models are complemented by OO-Method models that represents functional and persistence layers. OOWS generates the code corresponding to the user interaction layer, and OLIVANOVA [5], the industrial OO-Method implementation, generates the business logic and the persistence layers.

4 The approach in action

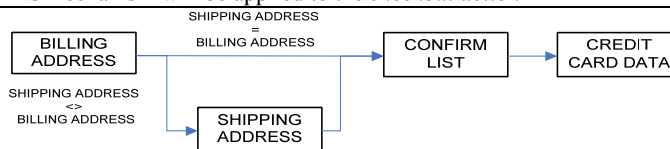
To illustrate the approach we will show concrete examples of usability requirements following the same basic notions commented in section 3.2. Functional requirements have already been dealt with in

[34], so for the sake of conciseness, we will stress how to deal with usability requirements and show only a couple of UIDs.

Assuming we have already developed the concepts of *product*, *category* and *list of products* in the E-commerce application, we are about to start a new sprint that concentrates in the *shopping cart*. We will show the approach in the context of improving the usability of the existing checkout process which is performed in a single web page. Specifically, we want to include a wizard to carry out the checkout process. To do so, we are going to use a FUF called Wizard. This FUF has a usability mechanism called Step by Step which has the goal of helping the user in complex tasks. We will go through the following steps to develop this requirement:

1. We extract the usability properties from the requirements guideline of the usability mechanism called Step by Step (Table 1). Those properties specify the service that will be executed at the end of the wizard; how we have to split the navigation concern; the description for each step; how the information will be displayed in each step, and whether or not each step will inform about the number of remaining steps. We need to refactor the current mockups to show what we expect from a UI perspective.

Table 1. Usability properties for Step by Step

Step by Step	
Property	Value specified by the analyst in the checkout example
Service selection	This mechanism will be applied to the <i>checkout action</i>
Steps division	
Step description	Each step must contain a short description
Visual aspect	The user has specified the widgets to fill in each step
Remaining steps	The system must inform about the number of remaining steps

2. We capture the navigation between the different steps of the checkout process in a UID that will serve as a partial navigation model (see Figure 5.a), allowing the developer to implement the navigation aspect of the requirement. Then, we rework on the mockup of the checkout process by splitting it into several steps (Figure 5.b shows some resulting mockups for these steps). We add the necessary widgets as described in the table 1 to show the remaining steps and their descriptions (on each node).

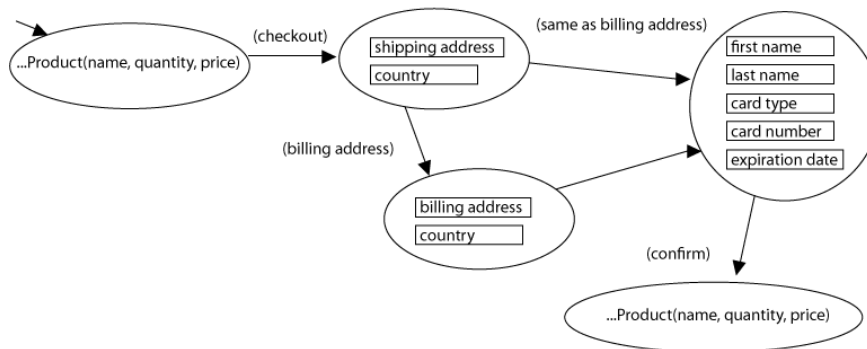


Figure 5a UID for checkout steps.

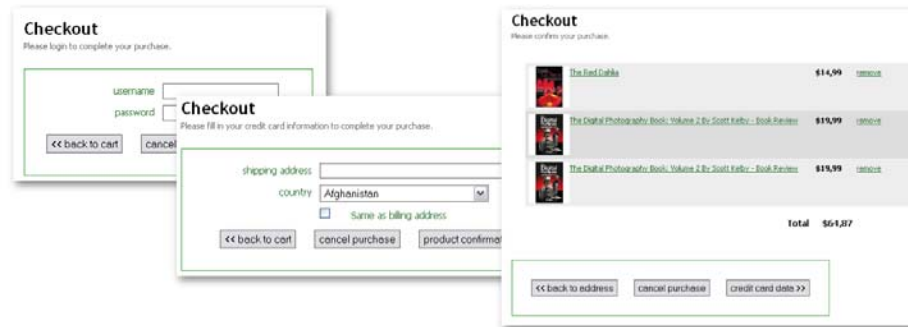


Figure 5b Sample mockups for checkout steps.

3. We refactor the existing checkout test so that it specifies the new process. We have to add the necessary asserts to validate the description, remaining steps, etc. Then, we run the test to check whether it is a new requirement and the application does not support it yet. We next show a test in Selenium Java notation:

```
public class CheckoutTestCase extends SeleneseTestCase {
    public void testSuccessfulCheckout() throws Exception {
(01) sel.open("file:///dev/bookstore/Mockups/books-list.html");
(02) sel.clickAndWait (
        "/ul[@id='products']/li[1]/div[1]/div[@id='prod-info']/a");
(03) sel.assertLocation("/cart*");
(04) assertEquals(
        "The Digital...",
        sel.getText("/ul[@id='selected-products']/li[1]/span[1]"));
(05) sel.clickAndWait("checkout");
(06) sel.assertLocation("/checkoutStepShippingAddress");
(07) assertEquals("3", sel.getText("//div[@id='remaining']"));
(08) assertEquals(
        "Shipping information",
        sel.getText("//div[@id='stepDescription']"));
(09) sel.type("shipping-address", "Calle 58");
(10) sel.select("country", "label=Argentina");
(11) sel.clickAndWait ("//input[@value='billing information>>']");
(12) sel.assertLocation("/checkoutStepBillingAddress");
(13) assertEquals("2", sel.getText("//div[@id='remaining']"));
(14) assertEquals(
        "Billing information",
        sel.getText("//div[@id='stepDescription']"));
(15) sel.type("billing-address", "Calle 48");
(16) sel.select("country", "label=Argentina");
(17) sel.clickAndWait ("//input[@value='product confirmation>>']");
(18) sel.assertLocation("/checkoutStepProductConfirmation");
    }
```

```

(19) assertEquals("1", sel.getText("//div[@id='remaining']"));
(20) assertEquals(
    "Product confirmation",
    sel.getText("//div[@id='stepDescription']"));
(21) assertEquals(
    "The Digital...",
    sel.getText("//ul[@id='selected-products']/li[1]/span[1]"));
(22) sel.clickAndWait("//input[@value='credit card data >>']");
(23) sel.assertLocation("/checkoutStepCreditCardData");
(24) assertEquals("0", sel.getText("//div[@id='remaining']"));
(25) assertEquals(
    "Credit card information",
    sel.getText("//div[@id='stepDescription']"));
(26) sel.type("first-na", "Esteban");      sel.type("last-na", "Robles");
(27) sel.type("card-number", "4246234673479");
(28) sel.select("exp-year", "label=2011");
(29) sel.select("exp-month", "label=Apr");
(30) sel.clickAndWait("//input[@value='confirmation >>']");
(31) sel.assertLocation("/checkoutSucceed");
(32) assertEquals(
    "Checkout succeeded",
    sel.getText("/div[@id='message']"));
}
}

```

The test opens the book list page (1) and adds an item to the shopping cart (2). Then we assert that the book has been added and proceed to the checkout process (3-5). Shipping information (6-11) and billing information (12-17) are filled and confirmed. Products are confirmed by asserting that product's name (18-22). Credit card data is filled (23-30) and then we confirm the process has succeeded by looking at the text displayed in a *div* element (31-32).

4. Since we are modelling the application with OOWS [13] we have to extend the navigation, domain and UI models to fulfil this new requirement. The Conceptual Model Compiler associated to OOWS is the responsible of creating the web application corresponding to the extended models. The strategy followed by it is out of the scope of this paper, but the reader will find the relevant details in [13].
5. We check that the application obtained in step 4 satisfies the requirements by running the whole test suite. If one test fails then we have to go back to step 4.
6. We get a new version of the application by integrating new changes with the current version of the model.

4.1 Handling usability requirements

As we have previously seen, usability requirements with functional implications have already been catalogued in the literature [22] as Functional Usability Features (FUF). These FUFs are derived from usability heuristics, rules and principles. In other words, FUF are functional requirements that improve particular usability attributes. Moreover, FUFs are divided into different specialised subtypes called

usability mechanisms. The definition of these mechanisms includes a set of guidelines to lead the analyst in the usability requirements capture. The guidelines are composed of questions that the analyst must ask to the user in order to extract usability requirements. We have used those guidelines to identify the usability properties that must be considered in the early stages of the software development process. Properties are the different configuration possibilities that a usability mechanism has to adapt itself to usability requirements. More details about our proposal to extract usability properties from usability mechanisms guidelines can be found in [30].

In Figure 6 we show a sketch to explain how we have extracted usability properties from FUFs. Each FUF is divided into several usability mechanisms. These mechanisms include a guideline to capture usability requirements. From those guidelines, we have extracted a set of properties. Grey boxes in Figure 6 represent existing elements in the literature, while white boxes represent a new contribution of our work.

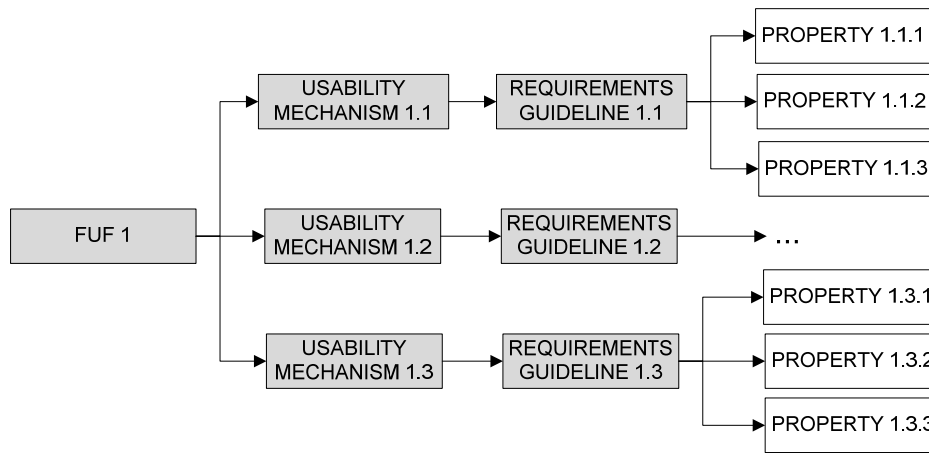


Figure 6 Division of FUF into properties

In this section, we show usability properties extracted from the definition of some usability mechanisms. Specifically, we focus on mechanisms related to Web applications environments. For each mechanism, we have specified its motivation according to existing works [22], the properties derived from it, and finally an overview of how those properties must be mapped to a concrete test suite. The test suite and the different scenarios are going to be explained in natural language because of the nature of FUFs: they are general and not concrete for a specific application. Later in the lab case presented in section 5 we are going to show how those scenarios are mapped in concrete software artefacts.

4.1.1 Favourites

The motivation of this mechanism is to let the users make a record of their points of interest, so that they can easily go back to them later. This mechanism allows the user to move freely through a way that is not directly supported by the structure of the system. The context of use of this mechanism is interfaces that the user visits frequently. Properties derived from the requirement guideline of Favourites are the following:

- Favourites' location: This property is used to specify where the list of favourites will be shown in the interface. For example, the list of favourites can be included in the main menu, in the main interface, in a specific window, etc.
- Num of items: This property specifies the maximum number of items listed in the favourites' area. Analyst must adapt this property according to the interface size and the user's requirements.

Test generation

The test suite for Favourites must include the following scenarios:

- Add favourite: Open the application. Navigate to a specific item that wants to be categorized as favourite. Add to favourite list. Check that the item is added to the favourite list.
- Navigate to favourite: Open the application. Identify as a user that already has favourites. Click on a favourite. Check that navigation has occurred to the specific item.
- Validate favourites' location: Open the application. Check the location of the favourites' area using an XPath expression.
- Validate number of items: Open the application. Add $n + 1$ (n is maximum) items as favourites. Check that favourite's area only contains n items.

4.1.2 Progress Feedback

The motivation of this usability mechanism is providing users with information related to the evolution of the requested services. The concept of service is defined as a processing unit that modifies the local state of an object according to [31]. The context of use for this mechanism is when a process interrupts the user interface for longer than two seconds. In that case, the system should show an animated indicator of how much progress has been made. Properties derived from the requirement guideline of Progress Feedback are the following:

- Service selection: This property is used to select which services will show the progress of their execution. Analyst must select the services that usually will spend more than two seconds in the execution.
- Visualization options: Analyst uses this property to decide how the progress will be shown to the user. This progress can be shown by means of a progress bar or by a list of completed services. Moreover, both types of visualization have several options. For example, the progress bar can be shown from right to left, from left to right, including the remaining time, including the remaining percentage, etc.

Test generation

The test suite for Progress Feedback must include the following scenarios:

- Check progress existence: Open the application. Execute a service that requires progress. Check the presence of the progress bar. Wait for the result to be loaded. Check that the result is properly loaded.
- Check services (optional in case those services must be shown): Open the application. Execute a service that requires progress. Check the presence of the progress bar and the list of completed services is shown. Check that the result is properly loaded.

4.1.3 Abort Operation

The motivation of this usability mechanism is to provide a way of cancelling the execution quickly. The functionality of Abort Operation consists in interrupting the processing and going back to the previous state. The context of use is when a process interrupts the user interface for longer than two seconds. In that context, the system should provide a mechanism to cancel the execution. Properties derived from requirements guideline of Abort Operation are the following:

- Service selection: This property is used to select which services can interrupt their execution when the user considers. The analyst must select those services whose execution will spend more than two seconds or services that can block the system.
- Visualization options: This property is used to specify how the abort option will be shown to the user. For example, this functionality can be accessed by means of a button in the main menu, a button in an emergent window together with the progress bar, a short cut, etc.

Test generation

The test suite for Abort must include the following scenarios:

- Check abort cancelled: Open the application. Execute a service that requires cancel. Click the cancel button. Wait for the page to be loaded. Check that a message saying that the operation has been cancelled is shown.
- Check abort bypassed: Open the application. Execute a service that requires cancel. Wait for the page to be loaded. Check that the service has been executed.

4.1.4 Warning

The motivation of this usability mechanism is to ask for user confirmation in case the service requested has irreversible consequences. The context of use is when a service that has serious consequences has been required by the user. The properties derived from the requirements guideline of Warning are:

- Service selection: The analyst must choose which services have irreversible consequences depending on the business logic.
- Condition: This property is used to specify when the warning message must be shown. The analyst must define this condition using stored information and data written in input fields.
- Visualization options: This property is used to specify how the warning message will be shown to the user. For example, the text format, whether the message will appear in an emergent window or not, etc.

Test generation

The test suite for Warning must include the following scenarios:

- Check warning bypassed: Open the application. Fill the necessary data to make the warning happened. Execute the service. Check the presence of the warning. Accept the warning and check that the service has been executed.
- Check warning cancelled: Open the application. Fill the necessary data to make the warning happened. Execute the service. Check the presence of the warning. Cancel the warning and check that the service has NOT been executed.

4.1.5 Structured Text Entry

The motivation of this usability mechanism is to guide the user when the system can only accept inputs from the user in an exact format. The context of use is widgets that require a mask to guarantee the correct format in the data entry. The properties derived from the requirements guideline of Structured Text Entry are:

- Widget selection: This property is used to choose the services that need a mask. The analyst must decide which input fields will have a mask according to the business logic.

- Regular expression: This property is used to specify the format that the widget requires. The format is specified by means of a logic expression.

Test generation

The test suite for Structured Text Entry must include the following scenarios:

- Check widget set invalid: Open the application. Navigate to the selected node. For each widget: Add invalid input to it. Try to execute the service. Check the presence of the error message.
- Check widget set valid: Open the application. Navigate to the selected node. For each widget: Add valid input to it. Try to execute the service. Check that the error is not present for the specific widget.

5 A Lab Case

As a lab case example we have selected a Web application of a library. This system is used to perform distantly the most frequent actions in a library by means of internet. More specifically, we focused our work on three functionalities: Opening the main window (home); looking for a specific book; renewing the loan. Figure 7 represents the OOWS model for the contexts of those three functionalities. Following, we are going to explain the usability properties that the system must support and the tests that we have defined to guarantee those properties. In order to simplify how the examples are shown, we are not going to show each test scenario; only the most significant ones. We have divided the explanation into usability mechanisms, and for each usability mechanism we show a simplified version of the final user interface where the value of the usability properties can be seen applied in the system.

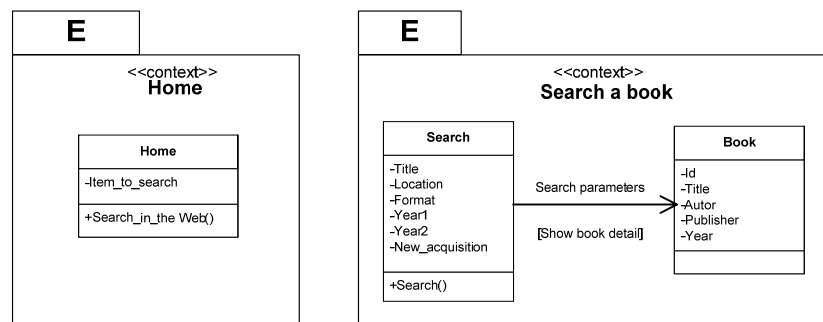


Figure 7a OOWS model for home and search a book

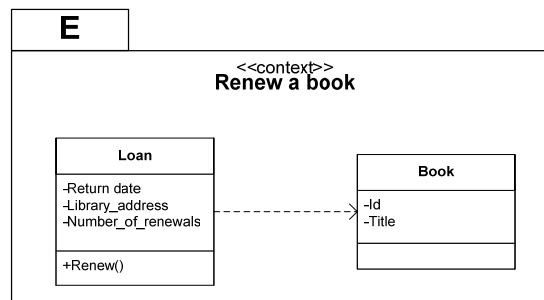


Figure 7b OOWS model for renew a book

In some cases, applying the usability mechanisms lead to changes not only in UI but in all three OOWS models, for example forcing us to change or grow the application's navigation structure. In such cases, the generated tests are again used as guide for the development through the changes in the navigation and domain models.

5.1 Favourites

According to the user requirements, the Web application should include in the main window a list of the most used interfaces. Therefore, the user can visit those interfaces directly from the main window, doing the user's work more efficient. The values for the usability properties of Favourites are shown in Table 2

Table 2. Usability properties for Favourites

Favourites	
Property	Value specified by the analyst in the library case study
Favourites' location	In the main window, in the right bottom
Num of items	Three items

Figure 8 shows a mockup of the main window where the list of favourites appears in the right bottom (*pages most visited*).

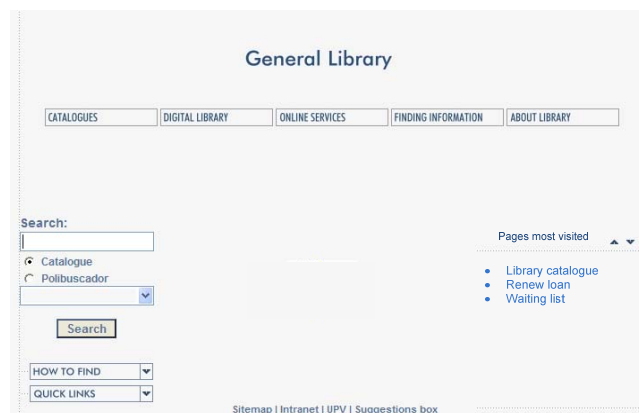


Figure 8 Mockup of the main window with favourites list

Test generation

Add favourite

```
Open("http://library.upv.es") //Open the application
ClickAndWaitPageToLoad("id=recomm1") //Open the first recommendation
AssertElementPresent("id=favourite0") //Assert that the recommendation has been
added
```

Navigate to favourite

```
Open("http://library.upv.es") //Open the application
Type("id=username", "Pablo") //Authenticate with an existing user
Type("id=password", "apasd")
```

```

ClickAndWaitPageToLoad ("id= login")
ClickAndWaitPageToLoad ("id= favourite0") //Navigate to a favourite
AssertLocation("book.asp") //Assert that navigation has occurred

```

Validate Num or items

```

Open("http://library.upv.es") //Open the application
For (I in 1..4) {
  ClickAndWaitPageToLoad("id=recomm1") //Open the first recommendation
  ClickAndWaitPageToLoad ("id=home") //Go back to home
}
//Assert that 3 elements are shown
For (I in 1..3) {
  AssertElementPresent ("id=favourite" + I)
}
AssertElementNotPresent("id=favourite4") //Assert that the 4th element is not shown

```

5.2 Progress Feedback

The process of looking for a specific book can take several seconds. According to usability requirements, the system should inform the user that the search service is in progress and how much time the user must wait until the search finishes. The values for the usability properties of Progress feedback are shown in Table 3.

Table 3. Usability properties for Progress Feedback

Progress Feedback	
Property	Value specified by the analyst in the library case study
Service selection	The search service
Visualization options	The system will show a progress bar in an emergent window where the user can see the percentage of the task that has been done. The progress will be drawn from left to right

Figure 9 Mockup of the window for searching a book

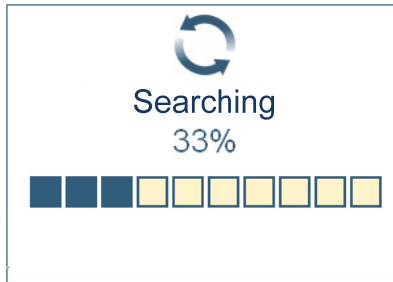


Figure 10 Mockup of a progress bar

Figure 9 shows a mockup where the user can look for a book. In that example, the user wants to look for books of software engineering. When the user presses the search button, a progress bar like Figure 10 will appear. This progress bar shows the percentage of finished task.

Test generation

Check progress existence

```
Open("http://library.upv.es") //Open the application
ClickAndWaitPageToLoad ("id=search") //Go to search
Type("id=searchField", "Development approaches");
ClickAndWaitPageToLoad ("id=doSearch") //Search
AssertElementPresent("id=pBar") //Check that progress bar is shown
WaitPageToLoad(30000)
AssertLocation("searchResults") //Check that search has occurred
AssertTextPresent("Development approaches")
```

5.3 Abort Operation

Once the search has been triggered, the user should be able to cancel this search. Sometimes the search may be too long or the user may have made a mistake triggering the search service. Therefore, the Abort Operation is a requirement for the library Web application. The values for the usability properties of Abort Operation are shown in Table 4.

Table 4. Usability properties for Abort Operation

Abort Operation	
Property	Value specified by the analyst in the library case study
Service selection	The search service
Visualization options	The cancel button will be shown in the emergent window together with the progress bar

Figure 11 shows a mockup for the Abort Operation. This usability mechanism has been implemented by means of a cancel button added to the progress bar showed in Figure 10. The user can abort the search service pressing the cancel button.

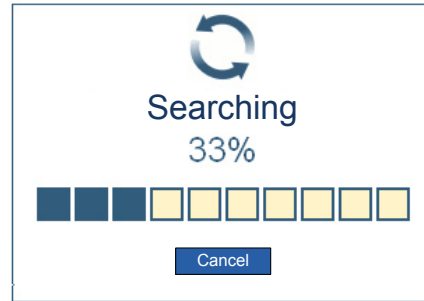


Figure 11 Mockup of a progress bar with abort operation

Test generation

Check abort cancelled

```

Open("http://library.upv.es") //Open the application
ClickAndWaitPageToLoad ("id=search") //Go to search
Type("id=searchField", "Development approaches");
ClickAndWaitPageToLoad ("id=doSearch") //Search
AssertElementPresent("id=pBar") //Check that progress bar is shown
ClickAndWaitPageToLoad ("id=cancel") //Search
AssertLocation("search") //Check that search has been cancelled

```

5.4 Warning

The Web application allows users to renew the loan of a book if there is none in waiting list to get that book. The renewal service let the user have the book one week more and this service can be executed only three times by loan. The renewal service cannot be undone; therefore the execution of the service has irreversible consequences. According to usability requirements, the system must warn the user about the consequences of the renewal service before its execution. The values for the usability properties of Warning are shown in Table 5.

Table 5. Usability properties for Warning

Warning	
Property	Value specified by the analyst in the library case study
Service selection	The renewal service
Condition	The warning message must be shown each time the user triggers the renewal service. Therefore the condition is “true” for every case.
Visualization options	The warning message will be shown in an emergent window, with arial font, size 10 and black colour.

Figure 12 shows a mockup to show a warning message when the user triggers the renewal service. Moreover informing the user about the consequences of the execution, the message asks for confirmation.

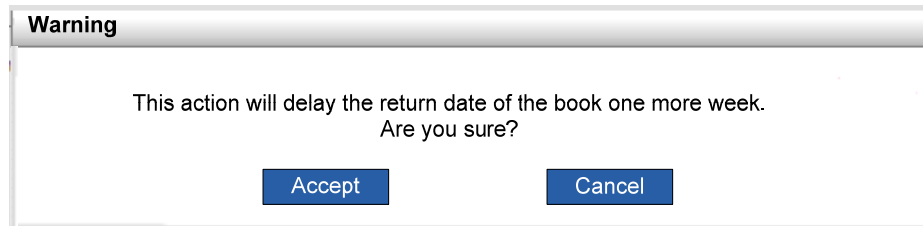


Figure 12 Mockup of a warning message

Test generation

Check warning bypassed

```

Open("http://library.upv.es") //Open the application
Type("id=username", "Pablo") //Authenticate with an existing user
Type("id=password", "apasd")
ClickAndWaitPageToLoad ("id= login")
ClickAndWaitPageToLoad ("id= myBorrowedBooks")
Click("id= renew0")
AssertElementPresent("id=dialog")
ClickAndWaitPageToLoad ("id=acceptWarning")
AssertTextPresent("Book renewal accepted")
AssertText("id=remainingDays0" , "8")

```

Check warning cancelled

```

Open("http://library.upv.es") //Open the application
Type("id=username", "Pablo") //Authenticate with an existing user
Type("id=password", "apasd")
ClickAndWaitPageToLoad ("id= login")
ClickAndWaitPageToLoad ("id= myBorrowedBooks")
Click("id= renew0")
AssertElementPresent("id=dialog")
ClickAndWaitPageToLoad ("id=cancelWarning")
AssertTextPresent("Book renewal cancelled")
AssertText("id=remainingDays0" , "1")

```

5.5 Structured Text Entry

In order to filter the search of a book, users can insert a rank of years in which the book was published. According to requirements, the years must be inserted with four digits. To avoid mistakes of users, the input widget for years must include a mask to guarantee that the user inserts four digits. The values for the usability properties of Structured Text Entry are shown in Table 6.

Table 6. Usability properties for Structured Text Entry

Structured Text Entry	
Property	Value specified by the analyst in the library case study
Widget selection	Two widgets where the rank of years must be inserted
Regular expression	The regular expression is #####, in other words, four integers.

Figure 9 shows a mockup of interface to look for a book. The widgets *published between* include a mask to guarantee that both years have four digits.

Test generation

Check widget set invalid

```
Open("http://library.upv.es") //Open the application
ClickAndWaitPageToLoad ("id=search") //Go to search
Type("id=publishedFrom", "20000")
Type("id=publishedTo", "2009")
ClickAndWaitPageToLoad ("id=doSearch") //Search
AssertTextPresent("Invalid from number. Must be 4 digits")
```

In order to pass all these tests, it is required to change some aspects of the OOWS conceptual model defined in Figure 7. The new OOWS models are shown in Figure 13. Usability Properties have been included by means of stereotypes (Progress bar, Cancel, Warning, Mask) and a new class (Favourites). This is where our approach provides the intended additional value by linking explicitly TDD with MSD: once tests are written (and the requirement still not implemented), the required changes are incorporated in the model (instead of in the program code), making true the metaphor of working at the conceptual modelling level for software production purposes, while fully exploiting the principles of the TDD approaches.

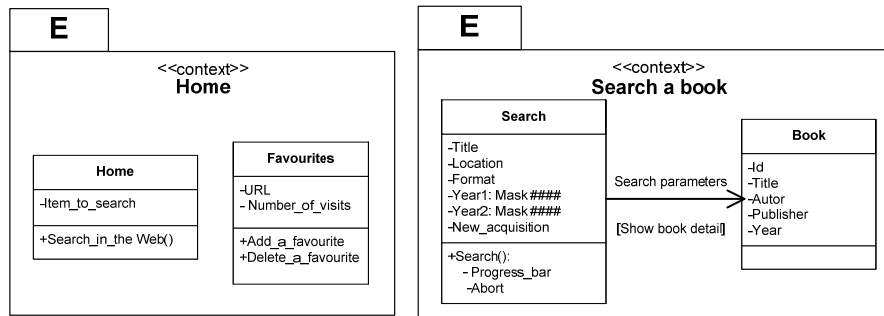


Figure 13a Modified OOWS model for home and search a book

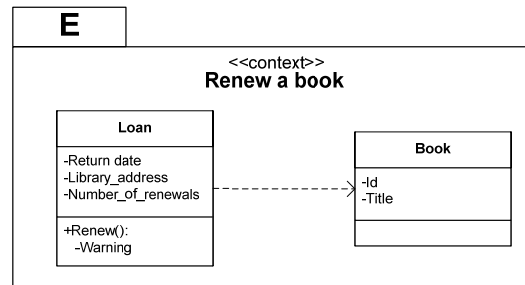


Figure 13b Modified OOWS model for renew a book

6 Concluding Remarks and Further Work

Recent studies have targeted the relationship between usability and functional requirements [2, 12]. We have presented a novel approach to include usability requirements strongly related to functionality

in a test driven, model-based development approach (MDSD). Usability properties are captured using a set of guidelines described in natural language. In order to fit this kind of requirements in the TDD cycle, we add tests that drive the development and check that the generated application is valid according to such requirements. The approach maintains the agile style while preserving an MDSD perspective, dealing with usability requirements in an incremental way. In order to exemplify our proposal, we have used a set of Functional Usability Features (FUF) for Web applications defined in the literature and we have explained how to define tests to validate those features in a specific MDSD method called OOWS.

Our proposal put together the advantages of agile methods and MDSD. On the one hand, all the software development process focuses on passing a set of tests defined with the help of the end user. That decreases possible misunderstanding between the end user and the analyst because the software can be validated quickly. On the other hand, the analyst concentrates all his/her efforts on building conceptual models, which are closer to the problem space than the implemented code. Additionally, a widely accepted software quality characteristic (usability) is incorporated to the proposed software production process from the requirements capture step. With all this work, it is our intention to demonstrate that:

- i) Agile and MDSD can be adequately combined to reinforce each other, focusing on their respective good properties from a methodological perspective. We have shown that there are no contradictions associated to their combined use.
- ii) Usability requirements can be incorporated to a MDSD method. Moreover, we have explained the advantages of dealing with usability from the early steps of the MDSD: less changes in the architecture design once the user sees the implemented system, usability can be included easily in the developing system by means of conceptual primitives.

We are currently working on several directions: First, we are working on an UID extension to easily derivate tests. As a proof of concept, we are developing a MDD tool that will simplify the process of UID construction and test generation. Second, we are doing some field experiences with usability requirements on RIA applications [10]. For this matter, we are analyzing how to validate those requirements in tests and where they should appear in the TDD cycle. Finally, in order to integrate all these features, we will extend the UID notation and tool to allow the specification of RIA properties.

Acknowledgements

This work has been developed with the support of MICINN under the project SESAMO (TIN2007-62894) and has been co-financed by ERDF. It also has the support of the Generalitat Valenciana by means of the ORCA project (PROMETEO/2009/015).

References

1. Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development. <http://www.agilemodeling.com>
2. Bass, L., Bonnie, J.: Linking usability to software architecture patterns through general scenarios. The journal of systems and software 66 (2003) 187-197
3. Beck, K.: Test Driven Development: By Example (Addison-Wesley Signature Series), 2002
4. Bryc, R.: Automatic Generation of High Coverage Usability Tests. Conference on Human Factors in Computing Systems (CHI), Doctoral Consortium. ACM, Portland, USA (2005) 1108-1109
5. CARE: www.care-t.com. Last visit: October 2009

6. Ceri, S., Fraternali, P., Bongio, A. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks and ISDN Systems*, 33(1-6), 137-157 June (2000).
7. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishing, London (2000).
8. Cysneiros, L.M., Kushniruk, A.: Bringing Usability to the Early Stages of Software Development. *International Requirements Engineering Conf. IEEE(2003)* 359- 360
9. Dinh-Trong, T.T., Ghosh, S., France, R.B.: A Systematic Approach to Generate Inputs to Test UML Design Models. *17th International Symposium on Software Reliability Engineering (2006)* 95-104
10. Duhl, J. Rich Internet Applications. A white paper sponsored by Macromedia and Intel, IDC Report, 2003
11. Estrada H., Martínez A., Pastor O. and Mylopoulos J., An empirical evaluation of the i* framework in a model-based software generation environment, *CAISE 2006, Springer LNCS 4001*, (2006) pp: 513-527.
12. Folmer, E., Bosch, J.: Architecting for usability: A Survey. *Journal of Systems and Software*, Vol. 70 (1) (2004) 61-78
13. Fons J., P.V., Albert M., and Pastor O: Development of Web Applications from Web Enhanced Conceptual Schemas. *ER 2003, Vol. 2813. LNCS. Springer (2003)* 232-245
14. Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
15. Hailpern, B., Tarr, P.: Model-Driven Development: the Good, the Bad, and the Ugly. *IBM Syst. J.* 45 (2006) 451-461
16. *IEEE Software*, vol. 24, no. 3, May/June 2007.
17. Gómez, J. and Cachero, C. 2003. OO-H Method: extending UML to model web interfaces. In *information Modeling For internet Applications*, P. van Bommel, Ed. IGI Publishing, Hershey, PA, 144-173.
18. Griffiths, R.: The Brighton Usability Pattern Collection. <http://www.cmis.brighton.ac.uk/research/patterns/home.html> (2002)
19. Jacobson, I, *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press, Addison-Wesley, 1992.
20. Jacobson, I., Booch, G. and Rumbaugh, J (1999). *The Unified Software Development Process*
21. Jeffries, R. E., Anderson, A., and Hendrickson, C. 2000 *Extreme Programming Installed*. Addison-Wesley Longman Publishing Co., Inc.
22. Juristo, N., Moreno, A.M., Sánchez, M.I.: Guidelines for Eliciting Usability Functionalities. *IEEE Transactions on Software Engineering*, Vol. 33 (2007) 744-758
23. Koch, N., Knapp, A., Zhang G., Baumeister, H.: UML-Based Web Engineering, An Approach Based On Standards. In *Web Engineering, Modelling and Implementing Web Applications*, 157-191. Springer (2008).
24. Lauesen, S.: Usability Requirements in a Tender Process. *Computer Human Interaction Conference*, 1998, Australia (1998) 114-121
25. Lawrence, B., Wiegers, K. and Ebert, C., The top risk of requirements engineering, *IEEE Software*, Vol. 18 (2001), pp: 62-63.

26. Maximilien, E. M. and Williams, L. 2003. Assessing test-driven development at IBM. In Proceedings of the 25th international Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 564-569.
27. MDA: <http://www.omg.org/mda> Last visit: October 2009.
28. Müller, M., Padberg, P. About the Return on Investment of Test-Driven Development. International Workshop on Economics-Driven (2003)
29. Nielsen, J.: Agile Usability: Best Practices for User Experience on Agile Development Projects. Nielsen Norman Group Report (2008)
30. Panach, J.I., España, S., Moreno, A., Pastor, Ó. Dealing with Usability in Model Transformation Technologies. ER 2008. Springer LNCS 5231, Barcelona (2008) 498-511
31. Pastor, O., Molina, J.: Model-Driven Architecture in Practice. Springer, Valencia (2007)
32. Perzel, K., Kane, D.: Usability Patterns for Applications on the World Wide Web. PloP'99 Conference (1999)
33. Rasmussen, J.: Introducing XP into Greenfield Projects: lessons learned. *IEEE Softw*, 20, 3 (May-June 2003) 21- 28
34. Robles Luna, E.; Grigera, J.; Rossi, G.: Bridging Test and Model Driven Approaches in Web Engineering. ICWE 2009.
35. Rossi, G., Schwabe, D.: Modeling and Implementing Web Applications using OOHDm. In Web Engineering, Modelling and Implementing Web Applications, 109-155. Springer (2008).
36. Selenium web application testing system. <http://seleniumhq.org/>
37. TestNG: <http://testng.org/> Last visit: November 2009
38. Tidwell, J.: Designing Interfaces. O'Reilly Media (2005)
39. UML: <http://www.uml.org/> Last visit: November 2009
40. Welie, M.v., Traetteberg, H.: Interaction Patterns in User Interfaces. 7th. Pattern Languages of Programs Conference, Illinois, USA (2000)
41. XML Path Language (XPath). <http://www.w3.org/TR/xpath>
42. Yu, E.: Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In: IEEE (ed.): IEEE Int. Symp. on Requirements Engineering (1997) 226-235
43. Watir: <http://watir.com/> Last visit: November 2009
44. Wiczorek, S., Stefanescu, A., Fritzsche, M., Schnitter, J.: Enhancing test driven development with model based testing and performance analysis. Testing: Academic and Industrial Conf Practice and Research Techniques, TAIC PART '08 (2008)82-86.
45. Zhang, Y.: Test-driven modeling for model-driven development. *IEEE Software* 21 (2004) 80-86

Capture and Evolution of Web requirements using WebSpec

*The content of this chapter corresponds with the following publication: **Robles Luna E.**, Garrigos I., Grigera J., Winckler M. *Capture and Evolution of Web requirements using WebSpec*. *Proceedings of 10th International Conference on Web Engineering (ICWE 2010)*. Vienna, Austria. Acceptance rate: 20%. Core C.*

In the previous chapter we have defined an approach for web application development that uses tests to drive the development and relies on models for the construction of the web application. In this chapter we present the requirement artefact used to specify requirements related with user interaction. We show its definition and use in the different activities of the development cycle. In the figure shown below, we details the activities presented in this chapter and how they are related with the WebTDD approach:

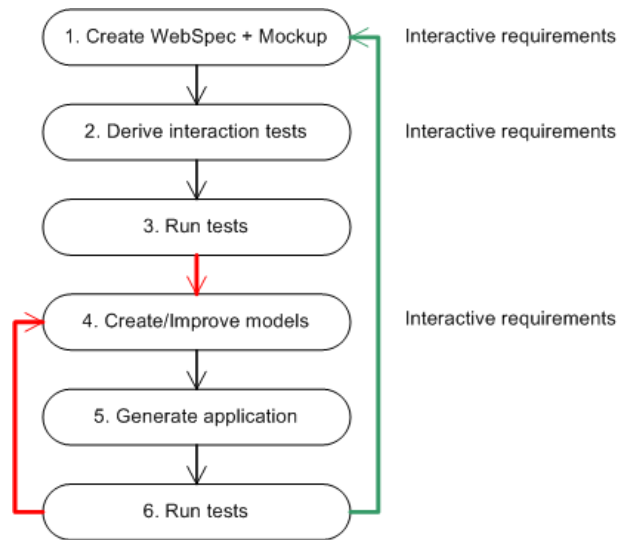


Fig. 4.1. The activities where Webspec is used to specify user interactive requirements

The content of this chapter is a paper published in the *International Conference of Web Engineering Support Systems (ICWE)*. ICWE aims at promoting research and scientific excellence on Web Engineering and at bringing together scientists and practitioners interested in technologies, methodologies, tools, and techniques used to develop and maintain Web-based applications.

Capture and Evolution of Web requirements using WebSpec

Esteban Robles Luna^{1,2}, Irene Garrigós³,
Julián Grigera¹, Marco Winckler⁴

¹LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
{esteban.robles, julian.grigera}@lifia.info.unlp.edu.ar

²Also at Conicet

³Lucentia Research Group, DLSI, University of Alicante, Spain
igarrigos@dlsi.ua.es

⁴IRIT, University Paul Sabatier, France
winckler@irit.fr

Abstract. Developing Web applications is a complex and time consuming process that involves different kind of people, ranging from customers to developers. Requirement artefacts play an important role as they are used by these people to perform their daily activities. However, state of the art in requirement management for Web applications disregards valuable features that tend to improve the development process, such as quick validation during elicitation, automatic requirement validation on the final application and useful change management support. To tackle these problems we introduce WebSpec, a requirement artefact for specifying interaction and navigation features in Web applications. We show its use through the development of an example application in the social networking area, and its implementation as an Eclipse plugin.

1 Introduction

It is usual to have multidisciplinary teams (including customers, analysts, developers, QA staff, etc) involved in the development of real world Web applications, making it a complex and time consuming process. Moreover, requirements are susceptible of changing along the development cycle, so it is important to keep them updated and record their changes to reduce risks and time efforts. Many times, the success of a Web project relies on how Web requirements are captured and specified [16].

Several studies [16, 19] in industrial cases have shown the importance of requirements in Web application development. Requirements are generally described in informal documents (e.g. use cases [13]) that are shared by the different stakeholders of the project. However, Web applications tend to evolve in short periods of time [16] and sometimes not having a comprehensive way of handling requirement changes in coherent documents. Therefore, testing against the requirement specification is not feasible [19]. Furthermore, it is sometimes necessary to get deeper in the development or design phases so that customers start to understand their own needs [19].

In this context, capturing requirements should be efficient enough to accomplish the time constraint, without disregarding the interactive nature of Web applications. Therefore, requirement artefacts have to be easily understood and validated by stakeholders prior to the development, in order to avoid future wastes of time. Moreover, during the development process, the application has to be checked to validate that new requirements have been correctly implemented without “breaking” previous ones. Furthermore, requirement artefacts should help to maintain good quality standards during the development process, which are hard to keep with short time constraints.

In the context of model driven Web engineering approaches [22, 20, 14, 2, 11] the aforementioned concerns are not generally taken into account [7]. As a consequence, Web applications developed with these methodologies share some commonalities with the industrial cases, such as outdated requirements, unfeasibility to test against the requirements and unsuitably to handle fast evolution. Web requirements artefacts (e.g. user interaction diagrams [22], extended use cases [6], etc) capture important aspects of Web applications like navigation; however they are either used to document [13] or to derive the first version of the domain or navigation models [8, 10] and do not consider either evolution or validation (except WebRe [8] which provide test derivation from WebRe models) or even quick validation during the capture phase.

To tackle these problems we present WebSpec, a multi purpose requirement artefact used to capture navigation, interaction and UI (User Interface) features in Web applications. To improve the capturing phase, WebSpec can be used in conjunction with mockups to provide realistic UI simulations, hence improving requirement elicitation. Also, to allow quick requirements’ validation in the final application, WebSpec automatically derives a set of interaction tests. Finally, WebSpec enforces change management support which could be used to improve the development cycle by automating structural changes in the application. Summarizing, we show how to:

- Simulate the application using WebSpec and mockups to improve communication between the different stakeholders and reduce elicitation times.
- Derive tests from WebSpec diagrams to reduce requirement validation times.
- Capture requirement changes and use them to semi/automatically upgrade the application and maintain quality standards.

The rest of the paper is structured as follows: in Section 2 we present WebSpec, its concepts and syntax. In Section 3 we show how it is used in different activities in the development cycle by improving requirement’s elicitation, helping to automatically validate the requirements and managing their changes. Section 4 briefly shows WebSpec Eclipse plugin and describes its use in a real application. Section 5 presents related work and finally in Section 6 we conclude and present further work.

2 WebSpec: a DSL to capture interactive Web requirements

WebSpec is a DSL (Domain Specific Language) that allows specifying navigation, interaction and UI aspects in a more formal way than, for example, use cases. A WebSpec diagram has two key elements: *interactions* and *navigations* (Fig. 1).

An *interaction* (the counterpart of a Web page in the requirements stage) represents a point where the user can interact with the application by using its interface objects (widgets). Interactions have a name (unique per diagram) and may have widgets such

as: labels, list boxes, buttons, radio buttons, check boxes and panels. Labels define the content (information) shown by an interaction. Interactions are graphically represented with a rounded rectangle which contains the interaction's name and widgets. A WebSpec diagram must have a starting interaction represented with dashed lines.

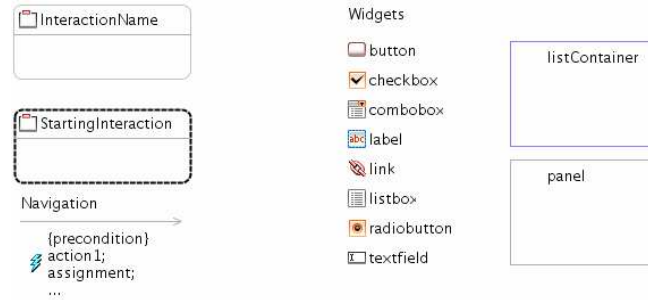


Figure 1. WebSpec's basic concepts

A mockup is a sketch of the “possible” application which generally represents UI elements. We can associate interactions with mockups and WebSpec widgets with their concrete UI elements in the mockup to improve the stakeholder's communication during the elicitation phase. There are several tools that could be used to create mockups, such as Balsamiq [1] or plain HTML. WebSpec allows using any of them as long as they provide a unique way to locate the interface elements.

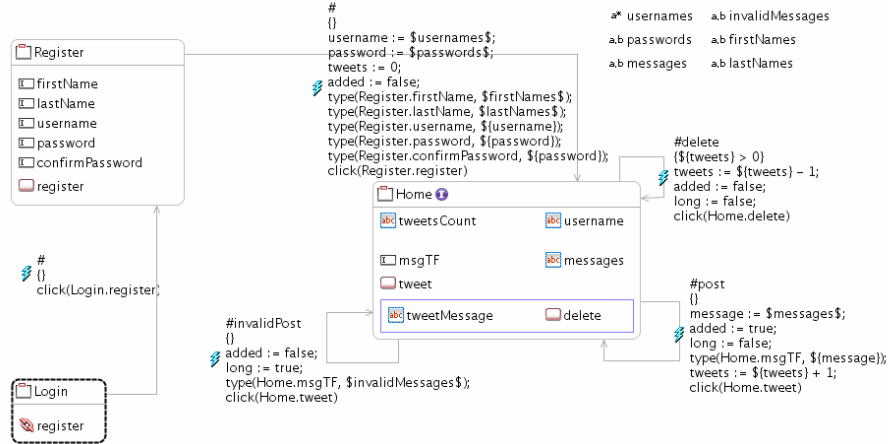


Figure 2. Tweet Webspec diagram

Invariants are Boolean predicates that must always hold. Every interaction has an invariant that specifies which properties must be satisfied (in case that we do not define one, it is assumed that the invariant is *true*). Fig. 2 shows a simplified diagram of a Twitter-like application that specifies the *post a message* (tweet) requirement and has 3 interactions named: Login, Register and Home. The Home interaction defines an invariant (marked with the I icon near the interaction's name): $Home.username = \{username\} \ \&\& \ Home.tweetsCount = \{tweets\} \ \&\& \ \{long\} \rightarrow Home.messages = \text{“Invalid message”}$ that states that the contents of the username label must be equal to

the username variable (denoted as $\$ \{variableName\}$) and the contents of the tweetsCount label must be equal to the tweets variable and if the long variable is true then the contents of the messages label must be equal to “Invalid message”.

A *navigation* from one interaction to another can be activated if its precondition holds by executing a sequence of actions such as: clicking a button, adding some text in a text field, etc. As well as invariants, preconditions can reference variables previously declared in the diagram. For example, the *delete* navigation (Fig. 2) has the precondition: $\$ \{tweets\} > 0$. Navigations are graphically represented in the WebSpec diagrams with gray arrows while its name, precondition and actions are displayed as labels over them. Actions are written in an intuitive DSL conforming to the syntax: $var := expr \mid actionName(arg1, \dots argn)$. Traditional hyperlink navigation is represented with no precondition (indeed, an always true precondition) and with only one action click (follow) a link widget (see Login to Register navigation in Fig 2). An example of a more complex sequence of actions is the *invalidPost* navigation (Fig. 2):

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

The first 2 sentences (1-2) assign constant values to variables. Then some text generated by the invalidMessages generator (denoted between \$) is typed in the msgTF text field (3) and finally the tweet button is clicked (4).

WebSpec allows specifying general properties like “an error must be shown if the user tries to post a message with more that 150 characters” using generators. Following the idea of QuickCheck [3], we extract the data used for specifying interaction requirements into generators. If a property in a WebSpec diagram holds, then it must hold for any element that could be generated by a generator. A generator is a function that can be called from navigation actions (e.g. \$invalidMessages\$) and generates data. For example, Fig. 2 has 6 generators: usernames, passwords, messages and invalidMessages, firstNames, lastNames. The invalidMessages generator generates strings with size > 150, so when that *invalidPost* navigation is activated, some invalid text will be typed and because the long variable will be true an error message must be display (recall the invariant of the Home interaction) in the messages label.

For those Web requirements that have strong hidden behaviour (not perceived from an interaction point of view, e.g. send an email), Webspec could be combined with simple notes over the diagram or by linking navigations with use cases or user stories. For example, if an email has to be sent when a user posts a message, we can easily add a note over the *post* navigation.

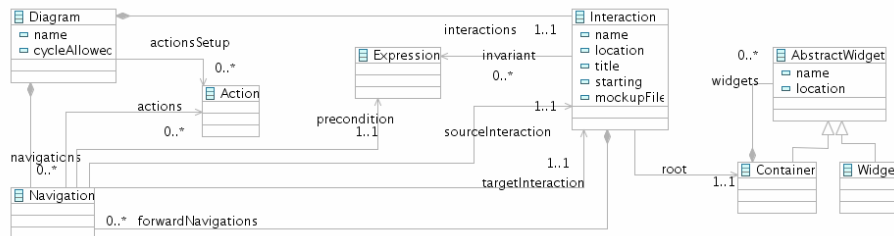


Figure 3. WebSpec simplified metamodel

Finally, WebSpec is formally defined in a metamodel (Fig. 3) that is used to improve the development process as shown in the following section. A diagram has a root object of the class Diagram which contains many Interaction and Navigation instances. An Interaction instance knows its name, forward navigations and associated mockup. A Navigation knows its source and target Interaction and the sequence of Action instances that triggers them. Finally, the interaction knows its root widget Container which can contain many AbstractWidget (Widget or Container) instances.

3 Using WebSpec along the development cycle

WebSpec allows specifying interaction requirements for Web applications at a conceptual level without imposing any particular development process. Notwithstanding, WebSpec diagrams can be used at different steps of the development cycle of Web applications. To illustrate this fact, we show in Fig. 4 how WebSpec can be used in the different activities of a test-driven approach like WebTDD [21] and in a methodology using a RUP [15] like process. Simulation (S in Fig. 4) can be used to share design options between stakeholders and validate their requirements in the requirements phase of both kind of processes. Tests generated from the diagrams (TG in Fig. 4) can be used to validate requirements against the final implementation when using a RUP style or to drive the development process in WebTDD. Changes during the development cycles are recorded (CR in Fig. 4) in the requirements phase of both. Finally, semi/automatic upgrades (CA in Fig. 4) using the previously recorded changes can be applied to the application in the development phase of WebTDD and RUP. In the following subsections we show how these features are supported in WebSpec.

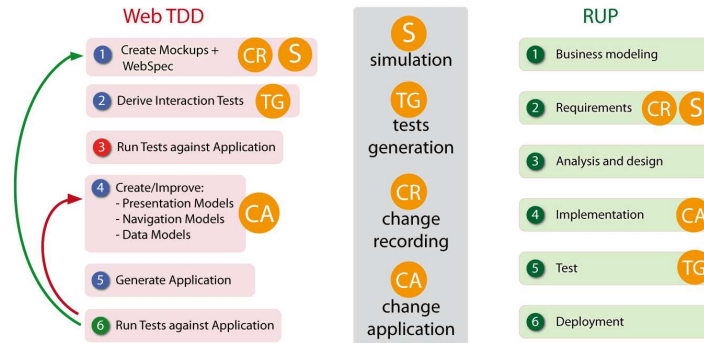


Figure 4. Using WebSpec in activities of different approaches

3.1 Simulating the application during requirements elicitation

With the aim of improving the requirement elicitation phase, WebSpec diagrams allow the simulation of the resulting application. Simulation is important to bridge the gap between the understanding of customers and designers about requirements thus getting real feedback from them.

Most requirement artefacts [13, 8, 1, 22] require some level of knowledge from customers to be fully understood, causing communication or understanding problems

during elicitation. WebSpec is not the exception; understanding a diagram may take some time and require some knowledge of WebSpec's concepts, e.g. variables and interactions. To ameliorate this scenario WebSpec provides some interesting features such as mockup association and formal specification which allows to formally simulating the application to improve the communication between stakeholders during elicitation. We say formally, because different from the simulation provided by tools such as Balsamiq [1], we not only show transitions between the pages but also execute real actions and provide descriptions of what would be the real output of the application directly over mockups. The descriptions provided are generated automatically from the WebSpec diagram and they are easy to understand because they are written in natural language. In this way, from every WebSpec diagram a set of simulations is automatically generated which could be used at any time by customers to understand the meaning of the diagram and suggest changes or improvements to the analyst.

The set of simulations is obtained following the different paths from the starting interaction of each WebSpec diagram. If the diagram has cycles (a path that contains more than one occurrence of an interaction) then we have to prune those paths to obtain finite paths. For example, in the Tweet Diagram (Fig. 2) we can obtain the following paths pruning them (as it is a cycled diagram) to a length of 5 interactions:

Login -> Register -> Home -> (post nav) Home -> (post nav) Home
Login -> Register -> Home -> (invalidPost nav) Home -> (post nav) Home
Login -> Register -> Home -> (post nav) Home -> (invalidPost nav) Home
Login -> Register -> Home -> (invalidPost nav) Home -> (invalidPost nav) Home
Login -> Register -> Home -> (post nav) Home -> (delete nav) Home

Each simulation is created following the sequence of interactions and navigations of the path and data is generated when a generator is referenced inside expressions. The path is transformed into a simulation model (not shown for space reasons) that specifies the simulation steps. A simplified version of the transformation algorithm is shown next:

```
(01) simulation := new Simulation();
(02) for (PathItem item : path.getItems()) {
(03)   if (item.isInteraction()) {
(04)     Interaction interaction = (Interaction) item;
(05)     simulation.openMockup(interaction.getMockup());
(06)     simulation.showPredicate(interaction.getInvariant());
(07)   } else {
(08)     Navigation navigation = (Navigation) item;
(09)     simulation.showPredicate(navigation.getPrecondition());
(10)     for (Action action : navigation.getActions()) {
(11)       simulation.simulateAction(action);
(12)     }
(13)   }
(14) }
```

Line 1 creates the simulation model. For every item (interaction or navigation) in the path (2): if it is an interaction (3) we show the mockup associated with it (5) and show the predicate of its invariant to describe which properties must hold (e.g. "The label should have the value 'John'") (6); if the item is a navigation, we show the precondition (9) and for every action we simulate it (10-12).

As an example of a simulation we next show a sequence of the simulation steps of the path: ***Login -> Register -> Home -> (post nav) Home -> (post nav) Home*** generated by

the algorithm. For space reasons, we can not show all the steps so we will describe the first 11 steps and show steps 8 through 11 (except step 10 which is equal to step 11 without the label) in Fig. 5.

```
(01) open("loginMockup.html");
(02) click("register", "the user clicks the register button");
(03) open("registerMockup.html");
(04) type("firstName", "John", "the user types 'John'");
(05) type("lastName", "Doe", "the user types 'Doe'");
(06) type("username", "john.doe", "the user types 'john.doe'");
(07) type("password", "aaa", "the user types 'aaa'");
(08) type("confirmPassword", "aaa", "the user types 'aaa'");
(09) click("register", "the user clicks the register button");
(10) open("homeMockup.html");
(11) showDescriptionNearTo("it should contain the text 'John'",
    "username");
```

Line 1 opens the first mockup. Line 2 clicks the register button and line 3 we simulate navigation by opening the mockup associated with the Register interaction. Lines 4-9 execute the actions to move from Register to Home interaction. Specifically, line 8 (Step 8 of Fig. 5) types 'aaa' to the confirm password field and line 9 (Step 9 of Fig. 5) clicks the register button. Line 10 simulates the navigation by opening the mockup associated with the Home interaction and finally line 11 (Step 11 of Fig. 5) shows the label with the condition that must be satisfied according to the filled information. Notice that the algorithm has to use generators in lines 4, 5, 6, 7, 8 to generate data according to the specification of Fig 2 (Register to Home navigation).

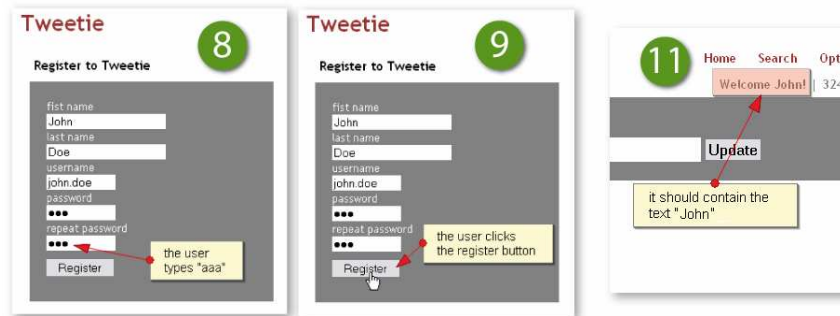


Figure 5. Simulation steps of the Tweet diagram

Once the requirements elicitation phase is completed we can automatically generate a set of tests that the application must pass as shown in the following subsection.

3.2 Automatic validation of requirements

New requirements must be validated to guarantee their correct implementation while previous ones still work as intended. However, it is hard to perform this task in short periods of time thus making it more important to keep requirements updated for the quality assurance team.

A well known way of validating requirements consists in running automated tests (that express the requirements) over the application. If one of these tests fails, then a requirement is not satisfied by the application. In particular, interaction tests play an

important role in industrial settings as they execute a set of actions in the same way a user would do on a real Web browser, thus their use is continuously growing [17]. However, in the Web engineering research area their use is recently appearing in approaches like WebTDD [21].

In a similar way we have created the simulations, we build a test suite (a set of test cases) from a WebSpec diagram by following the different paths from the starting interaction. To capture the basic concepts of tests, we have created a metamodel (Fig. 6) which is independent of the technology used. The metamodel contains the Test and TestSuite classes that conceptualize a test and a set of tests. A Test has a sequence of actions: assertions on interface objects or actions performed by the user over the application. Both cases are covered by the TestItem hierarchy.

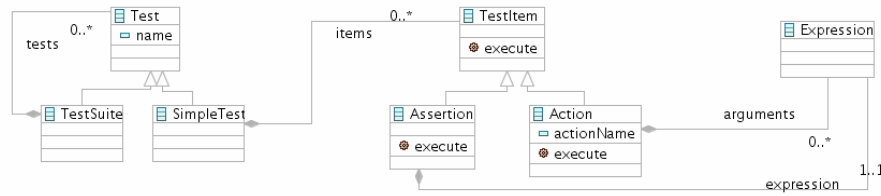


Figure 6. Test metamodel

To build the test suite, we transform each path into a SimpleTest (see Fig. 6) by executing the following simplified version of algorithm over each path. Similar to simulations, we will use generators to generate data according to the specification when an expression references it. The TestSuite is obtained by simple composition (see the composition relationship in the metamodel of Fig. 6) of the previous SimpleTest instances. More complex scenarios could be manually created by composing different Test suites into a bigger one. Once the TestSuite model is generated, we can translate it to a specific implementation framework such as Selenium [24].

```

(01) test := new SimpleTest();
(02) test.addItem(new OpenURL(applicationURL));
(03) for (PathItem item : path.getItems()) {
(04)   if (item.isInteraction()) {
(05)     Interaction interaction = (Interaction) item;
(06)     test.addItem(new Assert(interaction.getInvariant()));
(07)   } else {
(08)     Navigation navigation = (Navigation) item;
(09)     for (Action action : navigation.getActions()) {
(10)       test.addItem(new Execute(action));
(11)     }
(12)   }
(13) }

```

Line 1 creates the test model and line 2 generates the action to open the application. For each element in the path: if it is an interaction (4), we assert its invariant (6); if it is a navigation (8) we execute the actions that allow us to navigate from one interaction to another one (9-11).

To better illustrate these ideas, let us consider a specific path of the Tweet diagram: **Login** -> **Register** -> **Home** -> (post nav) **Home** -> (delete nav) **Home**. Applying the previous algorithm to the path and deriving a Selenium version of the test gives the next result:

```

(01) selenium.open("http://localhost:8080/index.html");
(02) selenium.click("id=register");
(03) selenium.waitForPageToLoad("30000");
(04) selenium.type("id=firstName", "John");
(05) selenium.type("id=lastName", "Doe");
(06) selenium.type("id=username", "john.doe");
(07) selenium.type("id=password", "wqe4yt24");
(08) selenium.type("id=confirmPassword", "wqe4yt24");
(09) selenium.click("id=register");
(10) selenium.waitForPageToLoad("30000");
(11) assertTrue((selenium.getText("id=username").equals("John"))
(12)    && (selenium.getText("id=tweetsCount").equals("0"))));
(13) selenium.type("id=tweetMessage", "@Office");
(14) selenium.click("id=tweet");
(15) selenium.waitForPageToLoad("30000");
(16) assertTrue((selenium.getText("id=username").equals("John"))
(17)    && (selenium.getText("id=tweetsCount").equals("1"))
(18) selenium.click("id=tweetDelete0");
(19) selenium.waitForPageToLoad("30000");
(20) assertTrue((selenium.getText("id=username").equals("John"))
(21)    && (selenium.getText("id=tweetsCount").equals("0"))));

```

Line 1 opens the application in the Web browser. Lines 2-3 click on the register link. Lines 4-10 fill the register information (first name, last name, username, password and confirm password) and clicks the register button. Lines 11-12 assert that the labels of the Home page have the values previously filled. Lines 13-15 post a new message to the wall. Lines 16-17 assert the new value that the labels must have after the post are valid. Lines 18-19 click on the delete button of the first message to delete the post. Finally, lines 20-21 assert the values of the labels after the delete operation.

As aforementioned, Web applications tend to change very fast, thus recording requirements changes is important to improve the development process. In the next subsection we show how requirement changes are captured in WebSpec.

3.3 Capturing requirement changes

Capturing requirements changes is an important feature to predict their impact in the application. Though some mature requirement artefacts [13] provide extensions to support change management, in the Web engineering field there are not many studies about how requirement changes can be captured and used to improve some part of the development process (see Sect. 5 for details).

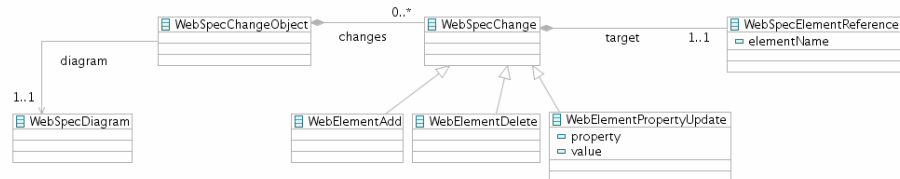


Figure 7. Change metamodel

In WebSpec, changes are recorded into change objects that group a set of changes. WebSpec can suffer different coarse grained changes, such as the addition or deletion of an interaction or navigation element. These elements can be modified too, by the addition or deletion of widgets to an interaction, changes in invariants, etc. As for navigations, we can add or delete preconditions, change their source, target, or the

actions that triggers them. All these types of possible changes have been represented in the metamodel of Fig. 7. When the user modifies the diagram, a change object is created and the sequence of changes is recorded as instances of these classes.

As an example, let us suppose we want to add a link between the Login interaction (Fig. 2) and a new TermsOfService interaction. The change in the diagram generates a new change object (Fig. 8) which has the following elements: a new interaction (TermsOfService), a new navigation (Login -> TermsOfService), a new link (tosLink) and a new label (the description of the terms of service). To take advantage of capturing changes, we show in the following subsection how to use WebSpec change objects to semi/automatically upgrade the application.

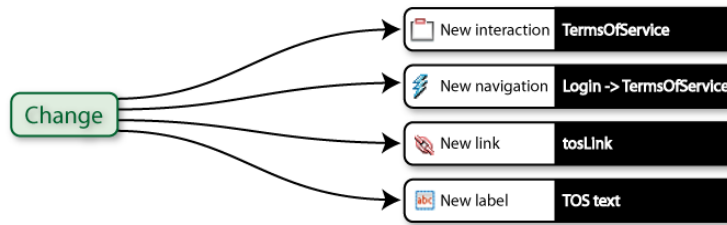


Figure 8. Change object representing the new Terms of Service functionality

3.4 Using requirement changes to evolve the application

Though handling requirement changes serves for multiple useful purposes, we will focus on how to semi automatically upgrade the application using them. Since change objects represent changes at the WebSpec level, we decouple the process of upgrading the application by providing different effect handlers. An effect handler is a component responsible of mapping the changes in the diagrams to a concrete technology and storing the trace links between the WebSpec elements and the technology ones. For example, a WebSpec diagram generates a change that can be applied with different effect handlers depending on the underlying technology: Seaside [23], GWT [12], WebRatio [25], etc. Seaside and GWT effect handlers will create/update methods and classes but WebRatio effect handler will produce model transformations in order to update the models.

As an example of the use of effect handlers, we next show how to use the change object of the previous subsection to upgrade the application. We assume that the application is developed with Seaside, so we use the Seaside effect handler.

The effect handler “reads” the change object and suggests actions to the developer. The first change (add the TermsOfService interaction) suggests to create a new class (WATermsOfService) that extends the base class of the Seaside framework (WALayoutPane) (see row 1 of Fig. 9). The developer accepts the proposal and continues with the next change that represents the navigation from Login to TermsOfService interaction. This change refers to behavioral aspects that the effect handler does not handle yet, so it does not propose an action. The two remaining changes involve adding widgets to the interactions. The first one adds a link in the Login interaction; because the effect handler stores the trace link between the interaction and the implementation class, it suggests adding a new method that creates the link to the WALogin class

(Row 2). Finally, the effect handler suggests adding a new method to the `WATermsOfService` to create the new label (Row 3).

change	generated code
New interaction: TermsOfService	<pre> WALayoutPane subclass: #WATermsOfService instanceVariableNames: '' classVariableNames: '' poolDictionaries: '' category: 'Twitter' </pre>
New link at Login	<pre> createTermsOfService control control := SFAnchor link: [self termsLinkPressed]. control name: 'terms';label: 'terms'. ^ control </pre>
New label at TermsOfService	<pre> createParagraph control control := SFParagraph new. control name: 'paragraph';contents: 'These Terms of Service (...) Terms. By accessing or using the Services you agree to be bound by these Terms.'. ^ control </pre>

Figure 9. Semi/automatic upgrades using the Seaside effect handler

4 Implementation

WebSpec has been implemented as an Eclipse plugin using EMF and GMF technologies. The plugin allows the creation of diagrams and the association of interactions with HTML mockups inside the environment. Simulations are implemented using a small extension to the Selenium framework, and JUnit selenium tests are automatically generated from diagrams. Finally, changes are recorded and stored into XML files that could be read by different effect handlers. We have implemented effect handlers for Seaside and GWT. Fig. 10 shows a screenshot of the WebSpec Eclipse plugin.

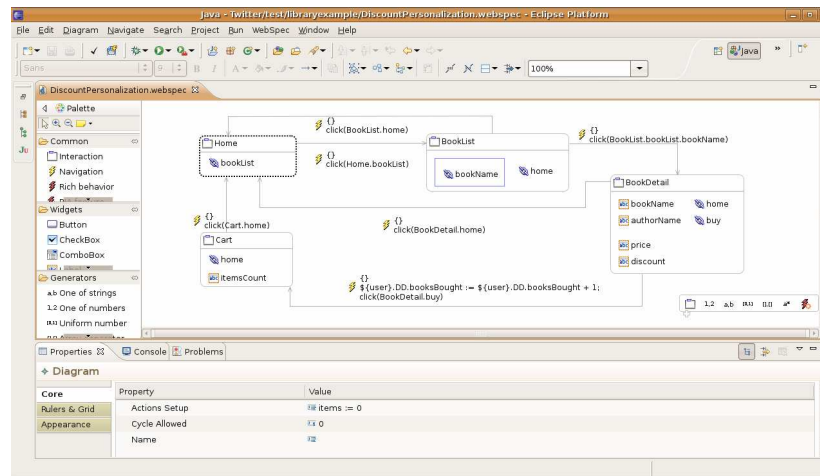


Figure 10. Webspec Eclipse plugin

Using the plugin and following the WebTDD approach, we have successfully implemented a complete application for the Post-graduate area of the College of Medi-

cine in the University of La Plata. We have used GWT, Spring and Hibernate as base technologies for the development process and actively used the generated tests to check that the application satisfies the requirements in an incremental way. Simulation was used for improving the elicitation of requirements and change objects allowed automating the creation of the structural UI classes of the application.

5 Related work

In the context of Web Engineering, the specification of interaction requirements is a complex task due to some unique characteristics of Web applications such as the need to represent the navigation in information spaces, the need of describing technical constraints related to the information flow (e.g. session management), the rapid evolution of requirements, sensitive communication among developers and the participation of customers in the development process (e.g. marketing experts, editorial board, etc) [26]. In the last years, a large variety of model-based artefacts have been employed to capture Web requirements like UML use cases and sequence diagrams [4], User Interaction Diagrams [22], task models [27], and navigation models [11]. It is also worthy noting a widespread use of paper-based mockups to capture requirements related to the user interface of Web applications [9] which has lead to the development of advanced tools for sketching and storyboarding the user interface of Web applications such as Denim [18] and Balsamiq [1].

Concept		Artefacts used for representing requirements				
Behaviour	Navigation	Dependencies between UC	Dependencies between tasks	Navigation	Navigation arrows	Arrows
	Process	Use cases	Tasks,	WebProcess	WebSpec diagram	-
	User interaction	Functional requirements	Interactive tasks	User transaction	Action	-
	Constraints	OCL	Lotus operators	OCL	Precondition	Annotated text
	Information flow	-	Data transfer between tasks	Data transfer in user transaction	Data transfer between interactions	-
	Node / page	-	-	Node	Interactions / navigations	Prototype
Structure	Content	-	-	Content	Widgets	Widgets
	UI composition	-	-	-	Containers	Prototype
	User roles	Actor	Actor	WebUser	-	-

Table 1. Expressiveness power of requirement artefacts for Web applications

In Table 1 we compare the expressiveness power of some artefacts with respect to the concepts for representing Web requirements. As shown in the table, each artefact includes only part of the concepts required to express requirements of Web applications. For example, whilst use cases can be used to represent functional requirements, mockups (either paper-based or supported by tools) are more likely to capture and represent requirements related to the composition of the user interface. Task models allow expressing fine-grained functional requirements including navigation, user transactions and business processes. As can be seen, Web engineering methods have often included more than one artefact for capturing requirements; for example use cases are present in OOHDM [22] in combination with UIDS. Besides, use cases and activity diagrams, WebML [2] uses semi-structured textual descriptions to capture

additional information that can hardly be expressed using the former models. Similarly, UWE [14] proposes extended use cases, scenarios and glossaries for specifying requirements and WSDM [6] employs task models using concurrent task trees.

Currently, there is no consensus on which notation(s) should be used to capture and specify Web requirements. In order to provide a more uniform view on the coverage of requirements by each artefact, Escalona and Koch [8] have proposed a metamodel based on WebRE profiles [8]. Its main advantage is the automatic generation of conceptual models (content and navigation models) which automatically satisfy the requirements. Notwithstanding, some requirements such as detailed composition of the user interface and behaviour constraints cannot be fully described with this notation.

In another study, Escalona and Koch [7] have investigated how different Web engineering methods support the capture of requirements. They demonstrated that Web engineering methods do not pay equal attention to requirements. Some methods employ classical notations to deal with Web requirements or ignore this phase of the development process. It is interesting to notice that requirement artefacts might play several roles during the development process: they can act as communication tools (for elicitation requirements with clients), as elements for early specifications (that should be taken into account during implementation phases) and as checklists for assessing if the final implementation complies the initial requirements. Requirement checklists can indeed be employed in regression testing [28] for assessing in a longer term, the evolution of requirements expressed for a single application.

In [5] the authors have investigated the communication role of artefacts and they proposed MoLIC which acts as a kind of blueprint of the application and thus allowing professionals from multidisciplinary backgrounds to share the same understanding of the essence of the application. Other authors however, have investigated how to automate the generation of the system specification from the requirements specification; for example OOWS [20] which extends activity diagrams with the concept of interaction point to describe the interaction of the user with the system. It provides automatic generation of (only) navigation models from the tasks description by means of graph transformation rules. A-OOH [10] considers the i* framework in order to specify the requirements model which is goal-oriented. From this specification, the conceptual models (e.g. domain and navigation models) are generated by means of QVT transformations. Both OOWS and A-OOH approaches are examples of methods that specify requirements and provide code derivation; however the level of detail they provide make them unsuitable as communication tools with clients.

WebSpec supports features that tend to improve the development process when changes appear often and should be implemented fast, in comparison with the aforementioned requirement artefacts. It provides a means to describe several of the unique aspects of Web applications (such as navigation and information flow); when used in combination with mockups, it provides animated storyboards to improve the communication between stakeholders. Moreover, they contain enough information to support test generation independently of the development method. Finally, change support and effect handlers allow managing the fast evolution of the application.

6 Concluding Remarks and Further Work

In this paper we have presented WebSpec: a requirement artefact used to capture navigation, interaction and UI features in Web applications independently of the development process. WebSpec presents several advantages that help to improve the development cycle in short periods of time. We have shown its use in conjunction with mockups to provide a formal simulation of the final Web application, getting real feedback during the requirement elicitation phase. Furthermore, requirements expressed in WebSpec diagrams are easily validated due to the automatic derivation of interaction tests. Finally, it has been shown how keeping diagrams updated contributes to semi/automatically upgrade the application thus improving development times.

This work focuses on interactive requirements of Web applications. In the future we aim at exploring how WebSpec can be used in conjunction with other techniques for expressing non-interactive requirements such as accessibility and usability of Web applications. We are currently working on adding RIA expressiveness to WebSpec, so that RIA features (e.g. autocomplete, hover detail, etc) can be easily specified in the diagrams. Also, we aim to associate WebSpec diagrams to tasks, so we can monitor the progress of a development process. Finally, we are analyzing different alternatives to support the specification of requirements at the domain level which can be seamless integrated in WebSpec.

References

1. Balsamiq. Available at: <http://www.balsamiq.com/products/mockups>
2. Ceri, S., Fraternali, P., Bongio, A. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks and ISDN Systems*, 33(1-6), 137-157 June (2000).
3. Claessen K., Hughes J., "QuickCheck: a lightweight tool for random testing of Haskell programs", *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, vol. 35, pp. 268-279, September 2000.
4. Conallen, J., *Building Web Applications with UML*, Addison-Wesley, 2000, 300 p.
5. de Paula, M. G., da Silva, B. S., Barbosa, S. D. 2005. Using an interaction model as a resource for communication in design. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, USA, April 02-07, 2005), pp 1713-1716.
6. De Troyer, O., Casteleyn, S. Modeling Complex Processes for Web Applications using WSDM. In: *3rd Int. Workshop on Web-Oriented Software Technologies*. Oviedo, Spain (2003). At: <http://www.dsic.upv.es/~west/iwwost03/articles.htm>
7. Escalona, M.J., Koch, N.: Requirements engineering for web applications – a comparative study. *J. Web Eng.* 2(3) (2004) 193–212.
8. Escalona, M.J., Koch, N. Metamodeling Requirements of Web Systems. In *Proc. International Conference on Web Information System and Technologies (WEBIST 2006)*, INSTICC, 310--317, Setúbal, Portugal. 2006.
9. Flanagan, S. The Paper Version of the Web. In *Deeplinking*, available at: <http://deeplinking.net/paper-web/>
10. Garrigós, I., Mazón, J.N., Trujillo, J.: A Requirement Analysis Approach for Using i* in Web Engineering. In: *ICWE*. (2009), LNCS, 5648, 151-165.
11. Gómez, J., Cachero, C.: OO-H Method: extending UML to model web interfaces. In: van Bommel, P. (ed.) *Information Modeling For internet Applications*, pp. 144–173.

- IGI Publishing, Hershey (2003).
12. GWT. Available at: <http://code.google.com/webtoolkit/>
13. Jacobson, I., Object-Oriented Software Engineering: A Use Case Driven Approach, ACM Press/Addison-Wesley, 1992.
14. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-Based Web Engineering. An Approach Based On Standards. In: Web Engineering, Modelling and Implementing Web Applications, pp. 157–191. Springer, Heidelberg (2008).
15. Kruchten, P. 2003 The Rational Unified Process: an Introduction. 3. Addison-Wesley Longman Publishing Co., Inc.
16. McDonald A. and Welland R., Web Engineering in Practice, Proceedings of the Fourth WWW10 Workshop on Web Engineering, Page(s): 21-30, 1 May 2001.
17. Maximilien, E. M. and Williams, L. 2003. Assessing test-driven development at IBM. In Proceedings of the 25th international Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 564-569
18. Lin, J., Newman, M. W., Hong, J. I., and Landay, J. A. 2000. DENIM: finding a tighter fit between tools and practice for Web site design. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (The Hague, The Netherlands, April 01 - 06, 2000). CHI '2000. ACM, New York, NY, 510-517.
19. Lowe D, Web system requirements: an overview. Journal of Requirements Engineering pp 102–113. Springer-Verlag (2003).
20. Pastor, O., Abrahão, S., Fons, J.: An Object-Oriented Approach to Automate Web Applications Development. In: Bauknecht, K., Madria, S.K., Pernul, G. (eds.) EC-Web 2001. LNCS, vol. 2115, pp. 16–28. Springer, Heidelberg (2001).
21. Robles Luna, E., Grigera, J., and Rossi, G. 2009. Bridging Test and Model-Driven Approaches in Web Engineering. In Proceedings of the 9th international Conference on Web Engineering. Lecture Notes In Computer Science, vol. 5648. Springer-Verlag, Berlin, Heidelberg, 136-150.
22. Rossi, G., Schwabe, D.: Modeling and Implementing Web Applications using OOHDm. In: Web Engineering, Modelling and Implementing Web Applications, pp. 109–155. Springer, Heidelberg (2008).
23. Seaside. Available at: <http://www.seaside.st/>
24. Selenium web application testing system. Available at: <http://seleniumhq.org/>
25. The WebRatio Tool Suite. Available at: <http://www.webratio.com>.
26. Uden, L., Valderas, P., Pastor, O. An Activity-theory-based to analyse Web applications requirements. Information Research Vol. 13, N. 2. June 2008.
27. Winckler, M.; Vanderdonck, J. Towards a User-Centered Design of Web Applications based on a Task Model. In Proceedings of IWOST'2005. Porto, Portugal, June 12-13th 2005.
28. Zheng, J. 2005. In regression testing selection when source code is not available. In Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (Long Beach, CA, USA, November 07 - 11, 2005). ASE '05. ACM, New York, NY, 752-755. DOI= <http://doi.acm.org/10.1145/1101908.1101997>

Integrating an early phase of requirements to WebSpec

The content of this chapter corresponds with the following paper:
Robles Luna E., Garrigos I, Mazon J-N., Trujillo J., Rossi G.
An i-based Approach for Modeling and Tesing Web Requirements.*
Journal of Web Engineering (JWE). 2010. Impact factor: 0.531.
 JCR.

In the previous chapters we have presented the WebTDD approach and the WebSpec language and we have shown how can be used in conjunction to develop web applications. However, we have claimed that WebSpec could be used with other methodologies.

In this chapter we show how WebSpec could be integrated with a methodology that uses an early phase of requirements in which objectives and tasks of the system/organization are defined before capturing detailed requirements (like the ones captured by WebSpec).

Several times a formal language like i* is used to describe these relationships. In this chapter we show how to use WebSpec with i* to models to specify Web requirements. When we used both artefacts we can semi automatically validate that the objectives described in the i* model are correctly implemented in the application by using the automatic derivation of tests that WebSpec provides. In the figure below, we show the activities of our modified A-OOH approach where i* models and WebSpec are used in conjunction.

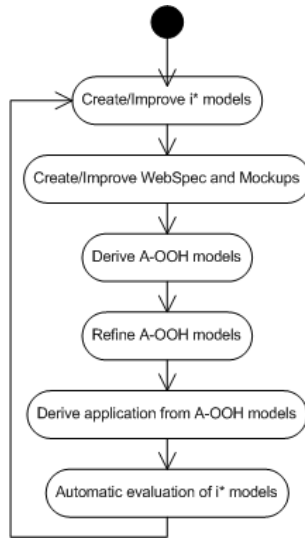


Fig. 5.1. Overview of our i*-based approach for Web application development which uses WebSpec diagrams to validate i* models

The content of this chapter is a paper published in the *Journal of Web Engineering (JWE)*. JWE aims to provide a forum for advancing the scientific state of knowledge in all areas of Web Engineering. JWE articles address significant issues and problems, and potential solutions.

AN I*-BASED APPROACH FOR MODELING AND TESTING WEB REQUIREMENTS

ESTEBAN ROBLES LUNA
LIFIA, UNLP, Argentina
erobles@lifia.info.unlp.edu.ar

IRENE GARRIGÓS, JOSE-NORBERTO MAZÓN, JUAN TRUJILLO
Lucentia Research group, University of Alicante, Spain
{igarrigos, jnmazon, jtrujillo}@dlsi.ua.es

GUSTAVO ROSSI
LIFIA, UNLP, Argentina
gustavo@lifia.info.unlp.edu.ar

Received June 1, 2010
Revised November 11, 2010

Web designers usually ignore how to model real user expectations and goals, mainly due to the large and heterogeneous audience of the Web. This fact leads to websites which are difficult to comprehend by visitors and complex to maintain by designers; these problems could be ameliorated if users are able to evaluate the application under development providing their feedback. To this aim, in this paper we present an approach for using the i* framework for modeling users' goals with mockups and WebSpec diagrams for detailing the specification of Web requirements, in such a way that the process of evaluating i* models for Web applications can be automated thus improving users' feedback during the development process. Also, as part of our development approach, we derive the domain and navigational models by defining a set of automatic transformations to a specific Web modeling method. Finally, we illustrate our approach with a case study to show its applicability and describe a prototype tool that supports the process.

Keywords: Requirement engineering, Web requirements, i*, goal evaluation

Communicated by: M. Gaedke, M. Grossniklaus, and O. Diaz

1 Introduction

In the last decade, the number and complexity of websites and the amount of information they offer is rapidly growing. In this context, introduction of Web design methods and methodologies [1, 2, 3, 4, 5] have provided mechanisms to develop complex Web applications in a systematic way. To better accommodate the individual user, personalization of websites has been also introduced and studied [6, 7, 8, 9].

However, traditionally, methodologies for Web engineering have not taken into serious consideration the requirement analysis phase. Actually, one of the main characteristics of Web applications is that they typically serve a large and heterogeneous audience in which i) everybody can access to the website and ii) each user has different needs, goals and preferences.

Importantly, this scenario hinders Web designers from considering users and current efforts for requirement analysis in Web engineering are rather focused on the system. Therefore, the

needs of the users are figured out by the designer and their user browsing experience may not be successful. Consequently, there may appear development and maintenance problems for designers, since costly, time-consuming and rather non-realistic mechanisms (e.g. surveys among visitors) should be developed to improve the already implemented website *a posteriori*, thus increasing the initial project budget.

To solve these drawbacks, in this paper, our aim is to model which are the expectations, intentions and goals of the users when they are browsing the site, and determining how they can affect the definition of a suitable Web design. Moreover, as Web applications have a strong emphasis in interaction, UI (User Interface) and navigation aspects, requirements related to these aspects should be also captured. The main benefit of our point of view is that the designer will be able to make decisions from the very beginning of the development phase. These decisions could affect the structure of the envisioned website in order to satisfy needs, goals, interests and preferences of each user or user type. Our work is inspired by agile software development [10] that states that the continuous evaluation of the application under development helps to gather feedback from users during development thus ameliorating some maintenance time-consuming tasks.

To this aim, we propose to use the *i** modeling framework [11, 12] for modeling requirements from the expectations and goals that users have (Fig. 1). The *i** framework is one of the most valuable approaches for analyzing stakeholders' goals and how the intended system would meet them. This framework is also very useful for reasoning about how stakeholders' goals contribute to the selection of different design alternatives. However, although *i** provides mechanisms to model stakeholders and relationships between them, it should be adapted for Web engineering, since the Web domain has special requirements that are not taken into account in traditional requirement analysis approaches. These requirements are related to the three main features of Web applications [13]: navigational structure, user interface and personalization capability.

Furthermore, because of the agile nature of Web applications there is a strong link between Web requirements and testing [14]. Specifically, defining test cases is needed to avoid erroneous implementations and deploying efficient Web applications meeting time constraints. Therefore, *i** is complemented in this paper with mockups^a and WebSpec diagrams [15]. We have chosen these artifacts because they help to agree on UI aspects, allow the specification of interactive Web requirements and provide automatic derivation of interactive tests to assess that the requirements are correctly implemented.

Once the requirements are specified, the next step is to obtain the conceptual models of the Web application (Fig. 1). To this aim, in this paper we also propose a set of QVT (Query/View/Transformation) rules [16] within a model-driven approach. In this way, designers will not have to create these models from scratch but they have a first tentative model satisfying the requirements specification and then they only have to refine these models, saving time and development effort. Though we use the A-OOH (Adaptive Object Oriented Hypermedia) [8] to illustrate our approach, any other Web engineering methodology could be used by only changing the QVT transformation rules. In the cases that the conceptual models of the Web methodology considered are based on UML, then the effort in updating the QVT

^aA mockup is a sketch of the “possible” application which generally represents UI elements and helps to agree on broad aspects of the Look and Feel of the application under development

transformations would be minimal, since A-OOH is UML-based. In the case of using another modeling language (different from UML), the modifications would be achievable, since all Web methodologies share similar basic concepts.

During the aforementioned process of model refinement (see Fig. 1), it is interesting to evaluate if the requirements and the goals specified in the i^* models are being satisfied. We propose that this model refinement is performed iteratively so that developers can tackle one requirement at a time, thus simplifying the process and in such a way that users can perceive the project's progress. From a user perspective it helps to ensure that time constraints are met and allow him to check (right after a task has been completed) if his expectations are actually satisfied in the developed application. In this way, the user will give its feedback during development like in agile development styles. In our approach i^* models are automatically evaluated by means of the interactive tests obtained from WebSpec diagrams. As a consequence, users could look at a tagged i^* model which states which goals are being satisfied by the application under development while developers refine the models.

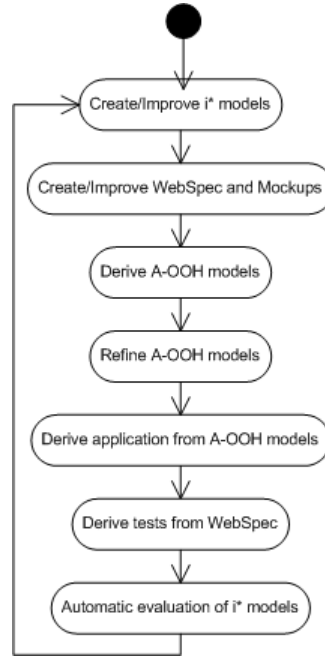


Fig. 1. Overview of our i^* -based approach for Web application development

In [17] we have already presented our requirement analysis approach for using i^* in Web engineering. In this paper we add a detailed analysis phase to specify interactive requirements in more depth. Also, from the elements used for requirement analysis, we add an automatic derivation of tests to evaluate the satisfiability of the requirements during the development process. Using our automatic evaluation method (Sect. 4) over the test results will serve to analyze if the application under development satisfies the goals described in the i^* model.

The remainder of this paper is structured as follows: our approach for requirement analysis in Web engineering is presented in Sect. 2. In Sect. 3 we show how to trace these requirements to a Web design model. Sect. 4 shows how the stakeholder's goals are iteratively validated

during the development cycle thus improving stakeholders' feedback. Sect. 5 describes an example of applying our approach. Sect. 6 describes related work. Finally, in Sect. 7, we present our conclusions and sketch some future work.

2 Modeling Requirements in Web Engineering

In this section, we present an approach for modeling requirements in Web engineering. Our approach begins capturing the needs and goals of the stakeholders (Sect. 2.1) in i^* models. Afterwards, a more detailed analysis is performed in order to capture interactive and UI aspects using a combination between mockups and WebSpec diagrams (Sect. 2.2). While we create the specifications in the diagrams, it is important to make explicit the relationships between the elements of the i^* model and the WebSpec diagrams in order to automatically evaluate the satisfiability of the goals during development (this issue will be addressed in Sect. 4 after presenting how to derive Web models in Sect. 3).

2.1 Modeling stakeholders' needs and goals

The development of Web applications involves different kind of stakeholders with different needs and goals. Interestingly, these stakeholders depend on each other to achieve their goals, perform several tasks or obtain some resource, e.g. *the Web administrator relies on new clients for obtaining data in order to create new accounts*. In the requirements engineering community, goal-oriented techniques, such as the i^* framework [11, 12], are useful for explicitly analyzing and modeling these relationships among multiple stakeholders (actors in the i^* notation). The i^* modeling framework provides mechanisms for representing (i) intentions of the stakeholders, i.e. their motivations and goals, (ii) dependencies between stakeholders to achieve their goals, and (iii) the (positive or negative) effects of these goals on each other in order to be able to select alternative designs for the system, thus maximizing goals fulfilment.

Next, we briefly describe an excerpt of the i^* framework which is relevant for the present work. For a further explanation, we refer the reader to [11, 12]. The i^* framework consists of two models: the strategic dependency (SD) model describes the dependency relationships (represented as \dashv) among various actors in an organizational context, and the strategic rationale (SR) model is used to describe actor's interests and concerns and how they might be addressed. The SR model (represented as \odot) provides a detailed way of modeling internal intentional elements and relationships of each actor (\bigcirc). Intentional elements are goals (\bigcirc), tasks (\diamond), resources (\square) and softgoals (∞). Intentional relationships are means-end links (\dashv) representing alternative ways for fulfilling goals; task-decomposition links (\dashv) representing the necessary elements for a task to be performed; or contribution links ($\xrightarrow[\text{hnt}]{\text{help}}$) in order to model how an intentional element contributes to the satisfaction or fulfillment of a softgoal.

Although i^* provides good mechanisms to model actors and relationships between them, it needs to be adapted to the Web engineering domain to reflect special Web requirements that are not taken into account in traditional requirement analysis approaches, thus being able to assure the traceability to Web design. Web functional requirements are related to three main features of Web applications [13] (besides of the non-functional requirements): navigational structure, user interface and personalization capability. Furthermore, the required data structures of the website should be specified as well as the required (internal) functionality

provided by the system. Therefore, in this paper, we use the taxonomy of Web requirements presented in [13]:

Content Requirements With this type of requirements the content that the website presents to its users is defined. Some examples might be: “book information” or “product categories”. Other kind of requirements may need to be related with one or more *content requirements*.

Service Requirements This type of requirement refers to the internal functionality the system should provide to its users. For instance: “register a new client”, “add product”, etc.

Navigational Requirements A Web system must also define the navigational paths available for the existing users. Some examples are: “consult products by category”, “consult shopping cart”, etc.

Layout Requirements Requirements can also define the visual interface for the users. For instance: “present a different style for teenagers”, etc.

Personalization Requirements We also consider personalization requirements in this approach. The designer can specify the desired personalization actions to be performed in the final website (e.g. “show recommendations based on interest”, “adapt font for visual impaired users”, etc.)

Non-Functional Requirements In our approach the designer can also model non-functional requirements. These kind of requirements are related to quality criteria that the intended Web system should achieve and that can be affected by other requirements. Some examples can be “good user experience”, “attract more users”, “efficiency”, etc.

Once this classification has been adopted, the i^* framework needs to be adapted to the Web domain. As aforementioned, our proposal is presented in the context of A-OOH (*Adaptive Object Oriented Hypermedia method*) Web engineering method [8], an extension of the OO-H modeling method [2], which includes the definition of adaptation strategies.

As this approach (A-OOH) is UML-compliant, we have used the extension mechanisms of UML to (i) define a profile for using i^* within UML; and (ii) extend this profile in order to adapt i^* to specific Web domain terminology. Therefore, new stereotypes have been added according to the different kind of Web requirements (see Fig. 2): *Navigational*, *Service*, *Personalization* and *Layout* stereotypes extend the *Task* stereotype and *Content* stereotype extends the *Resource* stereotype. It is worth noting that non-functional requirements can be modeled by directly using the softgoal stereotype.

Finally, several guidelines should be provided in order to support the designer in defining i^* models for the Web domain.

1. Determine the kind of users for the intended Web and model them as actors. The website is also considered as an actor. Dependencies among these actors must be modeled in an SD model.

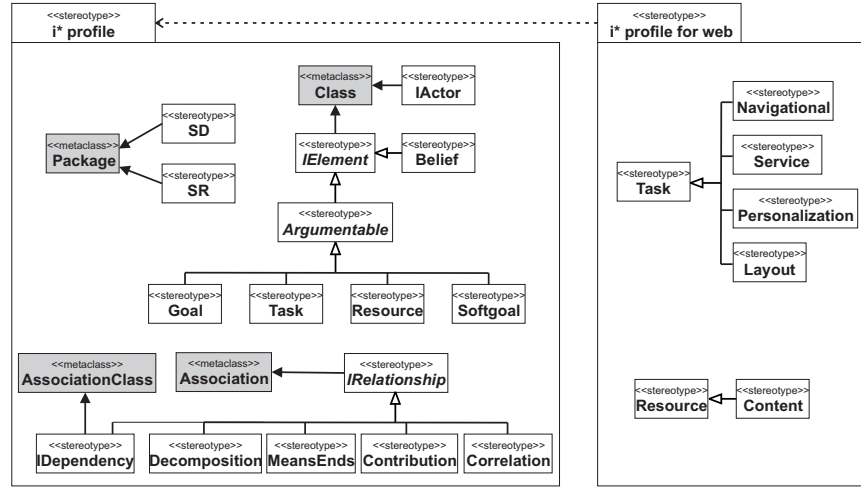


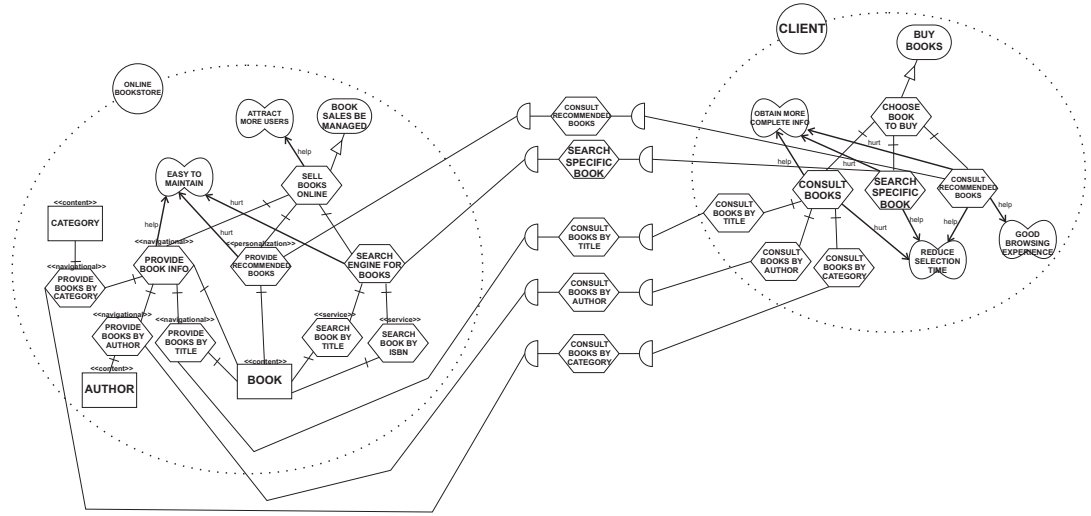
Fig. 2. Overview of the UML profiles for *i** modeling in the Web domain.

2. Define actors' intentions by using *i** techniques in an SR model [34]: modeling goals, softgoals, tasks and resources, and the relationships between them.
3. Define the *i** elements for the website actor and annotate tasks as navigational, service, personalization or layout requirements. Also, annotate resources as content requirements. It is worth noting that goals and softgoals should not be annotated.

To show the applicability of our approach, a case study is introduced. It is based on a company that sells books on-line. In this case study, a company would like to manage book sales via an online bookstore, thus attracting as many clients as possible.

A couple of actors are detected that depend on each other, namely “Client”, and “Online Bookstore”. A client depends on the online bookstore in order to “choose a book to buy”. These dependencies are modeled by an SD model (see Fig. 3). Once the actors have been modeled in an SD model, their intentions are specified in SR models.

The SR model of the online bookstore is shown in Fig. 3. The main goal of this actor is to “manage book sales”. To fulfill this goal the SR model specifies that a task should be performed: “books should be sold online”. We can see in the SR model that the first of the tasks affects positively the softgoal “attract more users”. Moreover, to complete this task three subtasks should be obtained: “provide book info” (which is a navigational requirement), “provide recommended books” (which is a personalization requirement), and “search engine for books”. We can observe that some of these tasks affect positively or negatively to the non-functional requirement “easy to maintain”: “Provide book information” is easy to maintain, unlike “provide recommended books” and “use a search engine for books”. The navigational requirement “provide book information” can be decomposed into several navigational requirements according to the criteria used to sort the data. These data is specified by means of content requirements: “book”, “author” and “category”. The personalization requirement “provide recommended books” is related to the content requirement “book” because it needs the book information to be fulfilled. The task “search engine for books” is decomposed into

Fig. 3. Modeling the intentional elements with *i**

a couple of service requirements: “search book by title” and “search book by ISBN”, which are also related to the content requirement “book”.

In the context of our case study the main goal of the client is to “buy books”. In order to fulfill it, the client should be able to perform the “choose a book to buy” task. The task “choose a book to buy” should be decomposed in several subtasks: “consult books”, “search specific book”, “consult recommended books”. These tasks can have positive or negative effects on some important softgoals. For example, “consult books” hurts the softgoal “reduce selection time”.

Note that tasks of the online bookstore actor have been stereotyped according with our profile. In this figure we can see that tasks “provide books by title”, “provide books by author”, “provide recommended books” have been stereotyped with *Navigational* and that the “search books by ISBN” task has been stereotyped with *Service*.

2.2 Modeling detailed interactive requirements

Because of the idiosyncrasy of Web applications, there are certain parameters that need to be considered when they are developed (e.g. time constraints, fast evolution, etc). To efficiently meet these parameters, it is crucial that interactive requirements are specified in more detail and also validated in the early stages of the development process. In order to perform this validation, requirements need to be analyzed in deeper detail, including navigation, UI and interactive aspects which are of paramount importance in the context of Web applications. To this aim, we propose to use mockups and WebSpec diagrams because mockups are widely used to agree on UI aspects and WebSpec provides automatic validation of requirements independently of the development approach used [15].

A WebSpec diagram specifies a set of scenarios that must be satisfied by the application. In order to specify scenarios, a WebSpec diagram is composed of elements of two different types that capture the concepts involved in interactive Web applications: interactions and navigations (Fig. 4).

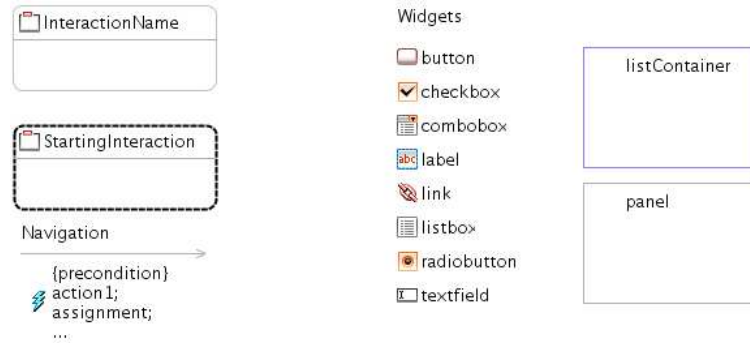


Fig. 4. WebSpec's basic concepts

An interaction (the counterpart of a Web page in the requirements stage) represents a point where the user can interact with the application by using its interface objects (widgets). Interactions have a name and may have widgets of different types such as: textfields, buttons, radiobuttons, panels, lists, etc. They are graphically represented with rounded rectangles containing the interaction's name and its widgets. The set of scenarios that the diagram specifies is obtained by following the different paths from a special interaction called “starting” denoted with dashed lines (Fig. 4).

To improve the communication between the different stakeholders, we can associate interactions with mockups and WebSpec widgets with their concrete UI elements to simulate the application [15]. There are several tools that could be used to create mockups, such as Balsamiq^b or plain HTML. WebSpec allows using any of them as long as they provide a unique way to locate the interface elements. As an example, Fig. 5 presents two mockups created with Balsamiq that show how the user will search books by title and see the results of that search. Notice that a Mockup has several labels with constant values which provide an example of the application the we are trying to develop.

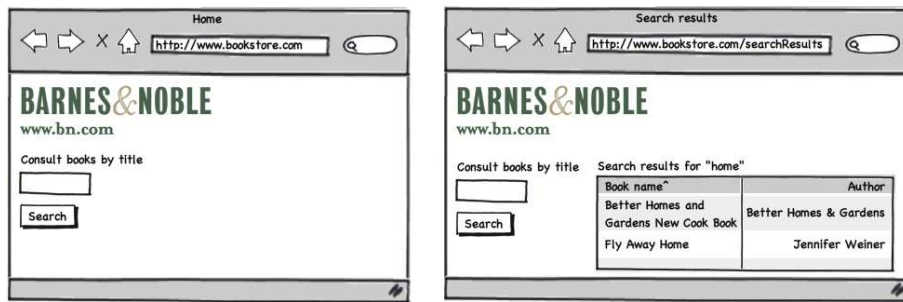


Fig. 5. “Consult books by title” mockup created with Balsamiq

To actually specify which properties must hold, we use invariants on each WebSpec interaction and in case that we do not define one, it is assumed that the invariant is true (it always hold independently of the interaction's state). Fig. 6 shows a simplified diagram of the “Consult books by title” of the Book store application. The diagram has 2 interactions

^b<http://www.balsamiq.com/products/mockups>

named: Home and SearchResults. The Home interaction represents the starting point of the scenario and, for the interest of this requirement, we assume that it must have 2 widgets: a search field and a search button. The SearchResults defines an invariant (marked with the I icon near the interaction’s name): that states: `SearchResults.title = “Search results for:” + ${productName}`. It means that the contents of the title label must be equal to the concatenation between “Search results for:” and the value of the `productName` variable (denoted as `${variableName}`).

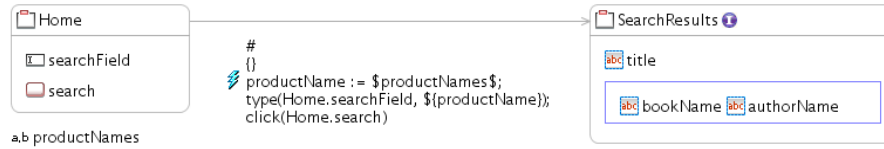


Fig. 6. “Consult books by title” WebSpec diagram

A navigation from one interaction to another can be activated if its precondition holds by executing a sequence of actions such as: clicking a button, adding some text in a text field, etc. As well as invariants, preconditions can reference variables previously declared in the diagram. Navigations are graphically represented in the WebSpec diagrams with gray arrows while its name, precondition and actions are displayed as labels over them. Actions are written in an intuitive DSL conforming to the syntax: `var := expr | actionName(arg1,... argn)`. For example the navigation from the Home interaction to the SearchResults performs three actions: first, it “generates” a `productName` (see [15] for further details of the use of generators), then this text is typed in the search field and finally the search button is clicked. We encourage the reader to look at [15] for further details about WebSpec.

After a diagram is created, a set of interaction tests can be derived from them [15]. These tests execute the actions and assert properties that are obtained from the navigations and invariants of the diagram which finally must be satisfied by the application. As the actions are performed directly over a Web browser, they are independent of the development approach used making the approach more appealing to be used within any Web engineering approach.

In order to evaluate the satisfiability of the *i** model according to the application under development, each of the tasks in the *i** model is specified in WebSpec diagrams that will specify the expected behavior of the application for fulfilling that task. It is worth noting that several WebSpec diagrams can be specified for each task as they represent different scenarios that must be satisfied. Consequently, tests derived from the diagrams are related with *i** tasks thus helping to analyze which tasks and goals are satisfied automatically.

3 Deriving Web Models

Once the requirements have been defined they can be used to derive the conceptual models for the website. Typically, Web design methods comprise three main models to define a Web application: a *Domain model*, in which the structure of the domain data is defined, a *Navigation model*, in which the structure and behavior of the navigation view over the domain data is defined, and finally a *Presentation model*, in which the layout of the generated hypermedia presentation is defined. In this work, for the sake of a better understanding, the focus is on the Domain and Navigation models.

this aim, OMG proposes the MOF 2.0 Query/View/Transformation (QVT) language [16], a standard approach for defining formal relations between MOF-compliant models.

QVT consists of two parts: declarative and imperative. The declarative part provides mechanisms to define relations that must hold between the model elements of a set of candidate models (source and target models). A set of these relations (or transformation rules) defines a transformation between models. The declarative part of QVT can be split into two layers according to the level of abstraction: the relational layer that provides graphical and textual notation for a declarative specification of relations, and the core layer that provides a simpler, but verbose, way of defining relations. The imperative part defines operational mappings that extend the declarative part with imperative implementations when it is difficult to provide a purely declarative specification of a relation.

In this paper, we focus on the relational layer of QVT. This layer supports the specification of relationships that must hold between MOF models by means of a relations language. A QVT relation (see Fig. 8) is defined by the following elements:

- **Two or more domains:** Each domain is a distinguished set of elements of a candidate model (source or target model). This set of elements (denoted by a `<<domain>>` label, see Fig. 8) must be matched in that model by means of patterns. A domain pattern can be considered as a template for elements, their properties and their associations that must be located, modified, or created in a candidate model in order to satisfy the relation. A relation between domains can be marked as check-only (labeled as C) or as enforced (labeled as E). When a relation is executed in the direction of a check-only domain, it is only checked if there exists a valid match in the model that satisfies the relationship (without modifying any model if the domains do not match); whereas for a domain that is enforced, when the domains do not match, model elements are created, deleted, or modified in the target model in order to satisfy the relationship. Moreover, for each domain the name of its underlying metamodel is specified (labels M1 and M2 in Fig. 8).
- **When clause:** This clause specifies the condition under which the relation needs to hold (i.e., it forms a precondition). This clause may contain arbitrary OCL (Object Constraint Language) [24] expressions in addition to the relation invocation expressions.
- **Where clause:** This clause specifies the condition that must be satisfied by all model elements participating in the relation (i.e., it forms a postcondition). This clause may also contain OCL expressions or relation invocation expressions.

Defining relations by using the QVT language has the following advantages:

1. QVT is a standard language.
2. Relations are formally specified, and transformation engines (e.g., Borland Together Architect^c, SmartQVT^d, mediniQVT^e, or ATL^f) can execute them automatically.

^c<http://www.borland.com/together>

^d<http://smartqvt.elibel.tm.fr>

^e<http://projects.ikv.de/qvt>

^f<http://www.eclipse.org/m2m/at1>

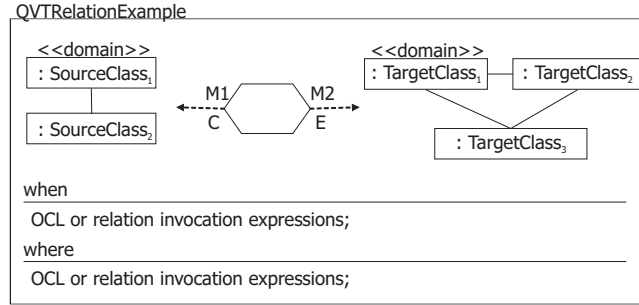


Fig. 8. Example of a QVT relation.

3. Relations can be easily integrated within any Web methodology (provided that meta-models are used).

Deriving the Domain model. The A-OOH DM is expressed as a UML-compliant class diagram. It encapsulates the structure and functionality required of the relevant concepts of the application and reflects the static part of the system. The main modeling elements of a class diagram are the classes (with their attributes and operations) and their relationships.

Table 1 summarizes how DM elements are mapped from the SR model. To derive a preliminary version of the DM we take into account two types of requirements defined in Sect. 2 content and service requirements. We have detected several patterns in the *i** models and we have used these patterns to define several transformation rules in QVT. Specifically, three transformation rules are defined in order to derive the DM from the SR model:

- *Content2DomainClass* By using this transformation rule, each content requirement is detected and derived into one class of the DM.
- *Navigation2Relationship* Preliminar relations into classes are derived from the relations among goals/tasks with attached resources by applying this rule. To generate the associations in the DM we have to detect a *navigational pattern* in the SR model of the *website* stakeholder. In Fig. 9(a) we can see that the *navigational pattern* consists of a navigational root requirement (i.e. task) which can contain one or more navigational requirements attached. Each of the navigational requirement can have attached a resource (i.e. content requirement). The classes mapped from the resources we find in such pattern will have an association relation between them. The QVT rule which describes this transformation is shown in Fig. 10.
- *Service2Operation* This transformation rule detects a *service pattern*, i.e. a service requirement with an attached content requirement in the SR model (see Fig. 9(b)). In this case each service requirement is transformed into one operation of the corresponding class (represented by the content requirement). In this QVT rule (shown in Fig. 11), a service pattern is detected and transformed into the corresponding elements in the target model.

Once the DM skeleton has been obtained it is left to the designer to refine it, who will also have to specify the most relevant attributes of the classes, identify the cardinalities and

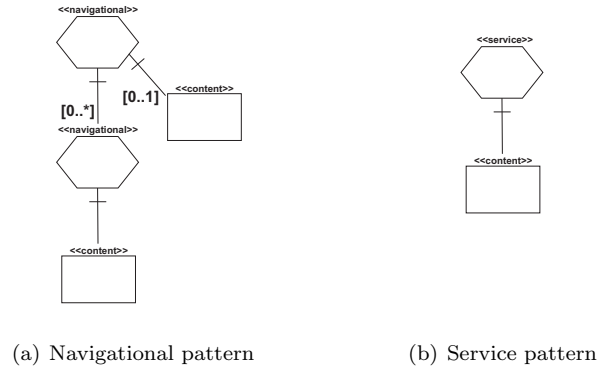


Fig. 9. Patterns.

define (if existing) the hierarchical relationships.

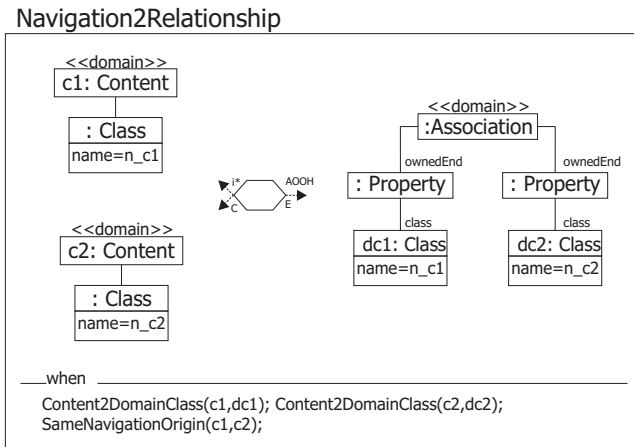


Fig. 10. QVT transformation rule for the navigation pattern

After the preliminary DM is created, a skeleton of the NM is also derived from the specified requirements. This diagram enriches the DM with navigation and interaction features. It is introduced next.

Deriving the Navigational model. The A-OOH Navigational model is composed of Navigational Nodes, and their relationships indicating the navigation paths the user can follow in the final website (Navigational Links).

There are three types of Nodes: (a) Navigational Classes (which are view of the domain classes), (b) Navigational Targets (which group the model elements which collaborate in the fulfillment of every navigation requirement of the user) and (c) Collections (which are (possible) hierarchical structures defined in Navigational Classes or Navigational Targets. The most common collection type is the C-collection (Classifier collection) that acts as an abstraction mechanism for the concept of menu grouping Navigational Links). Navigational

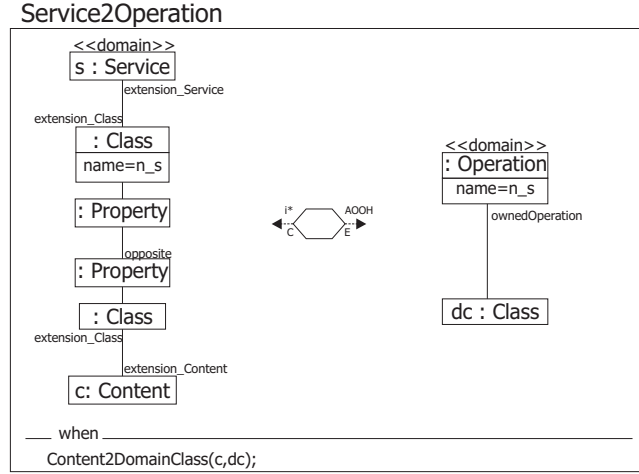


Fig. 11. QVT transformation rule for the service pattern in the DM

Table 1. Derivation of the Domain model

i* element	A-OOH element
Content Requirement	Class
Service Pattern	Operation
Navigational Pattern	Association between classes

Links (NL) define the navigational paths that the user can follow through the system. A-OOH defines two main types of links: Transversal links (which are defined between two navigational nodes) and Service Links (in this case navigation is performed to activate an operation which modifies the business logic and moreover implies the navigation to a node showing information when the execution of the service is finished).

To derive the NM we take into account the content requirements, service requirements and the navigation and personalization requirements. We also take into consideration the detected patterns (see Fig. 9) in order to develop several QVT transformation rules. In Tab. 2 we can see a summary showing how the different requirements are derived into elements of the NM. In the right part of Fig. 7 we can see the different transformation rules that are to be performed in order to derive a preliminary Navigation model. In this case we also define three transformation rules:

- *Nav&Pers2NavClass* By using this rule, a “home” navigational class is added to the model, which is a C-collection representing a Menu grouping navigational links. From each navigational and personalization requirement with an associated content requirement a navigational class (NC) is derived. From the “home” NC a transversal link is added to each of the generated NCs.
- *Navigation2TLink* This rule checks the *navigational pattern*, if it is detected, then a transversal link is added from the NC that represents the root navigational requirement to each of the NCs representing the associated navigational requirements.
- *Service2Service&SLink* Finally, the *service pattern* is checked by applying this transfor-

Table 2. Derivation of the Navigation model

i* element	A-OOH element
Navigation and Personalization Requirements	Navigational Class
Navigation Pattern	Transversal Links
Service pattern	Operation + Service Link with a target Navigational Class

mation rule. If a service pattern is found, then an operation to the class representing the resource is added and service link is created from each of the operations, with a target navigational class which shows the termination of the service execution. The QVT rule which describes this transformation is shown in Fig. 12.

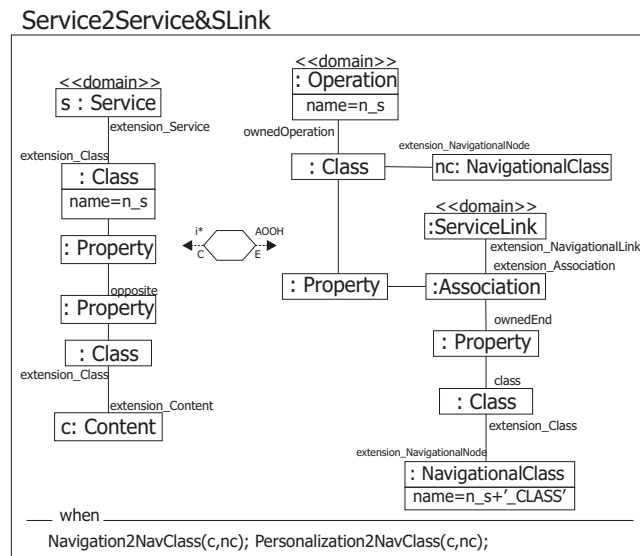


Fig. 12. QVT transformation rule for the service pattern in the NM

Finally, the derived NM could be refined by the designer in order to specify complementary elements for the desired navigation paths.

4 Automatic Goal Evaluation

In order to automatically evaluate whether the goals defined in the i^* model for Web applications are satisfied by the application, our approach extends the approach presented in [25] where manual or semi automatic evaluation of general i^* models is described. In that work, every task in the i^* models is tagged with one of these possible labels:

- Satisfied (✓): the element is satisfied.
- Partially Satisfied (✓): represents the presence of evidence which is sufficient to satisfy an element.

- Partially denied (X): represents the presence of negative evidence to satisfy an element.
- Denied (X): has evidence that the element is not satisfied.
- Conflict (X): indicates the presence of both positive and negative evidence of roughly the same strength.
- Unknown ($?$): represents the situation where there is evidence, but its effect is unknown.
- None: lack of any label.

Once an initial configuration of labels are set, an algorithm is executed to evaluate which goals are satisfied. This algorithm consists of propagating the labels that are given to the initial elements to the other elements. The algorithm is iterative and may require the intervention of the user if we can not decide the resulting label value.

Our evaluation of i^* models is done by using WebSpec diagrams to generate a set of test cases. In this way, an initial configuration of labels is obtained from test cases in order to verify and validate Web requirements. Our approach has a clear advantage: when users are involved in the development process of Web applications they want to know which goals are being satisfied while the application is under development (periodically, e.g. every hour). Our approach provides this automation without imposing any overhead to the development team.

In our i^* models for Web requirements, the actor that represents the Web application may have several tasks (of course, every task can be further decomposed in other tasks). For some of these tasks, WebSpec diagrams and mockups have been developed (by following the process shown in Sect. 2.2) so that we specify in more detail the behavior of the application and agree on broad aspects of the UI before the development begins. By using WebSpec features we automatically derive a set of interaction tests from the diagrams. These tests will assess if the application correctly implements the requirements that they express. Thus, there is a transitive relationship between Goals \leftrightarrow Tasks \leftrightarrow WebSpec diagrams \leftrightarrow Interaction tests. Indeed, if every test that is transitively related with a specific task is satisfied, then we can say that the task is satisfied too.

Our process for automatically giving initial labels to elements of our i^* models for Web requirements (assuming that we have a specific version of the application, an i^* model, and WebSpec diagrams and their associations) is as follows:

1. Each test (t_i) associated with a WebSpec diagram (WS) is run. If it passes then the edge (w_i) that links the diagram and the test as a weight of 1, otherwise 0.
2. A WebSpec diagram is $Y\%$ satisfied where $Y = \frac{\sum_{i=1}^z w_i}{z}$ and z is the size of tests that WS has.
3. A task (T) is $X\%$ done where $X = \sum_{j=1}^x h_j * Y\%$, x is the number of diagrams associated with T , h_j is a weight defined (only once per diagram) such that $\sum_{j=1}^x h_j = 1$.

Once the initial labels are set, we can reuse the i^* evaluation framework presented in [25] to automatically evaluate if the goals are satisfied based on the generated tags. However, tasks can now represent a percentile (instead of completely satisfied, partially satisfied, etc.)

thus we need to provide a mapping between our percentiles and the labels used to applied the algorithm. For example, we can define the following policy (note that the values for X, Y, and Z depend on the application under development and the actual characteristics of the project.):

- None: initial tag.
- None: if all the tests of a task have been failing since the beginning of the process.
- Satisfied: if the percentile of tests passed is $> X\%$.
- Partially Satisfied: if the percentile of the tests passed is between $X\%$ and $Y\%$.
- Partially Denied: if the percentile of the tests passed is between $Y\%$ and $Z\%$.
- Denied: if the percentile of tests passed is $< Z\%$.

One of the main advantages of our approach is that i^* models are evaluated for Web engineering in an objective and straightforward manner, thus avoiding the problem of deciding if a task has been performed or not. Also, it is lightweight and does not impose any overheads during development. In the following section we show our approach in action in our case of study.

5 Sample Application of our Approach

In the next subsections we explain our process described in Fig. 1 by means of an example based on the i^* model defined in Sect. 2.1 (see Fig. 3). We show how we specify in detail a specific requirement (Sect. 5.1), then a set of models is obtained (Sect. 5.2) which need to be refined. During the refinement process we can evaluate if our models satisfy the tasks by running the tests and evaluating their results (Sect. 5.3).

5.1 Detailed requirement specification

For the sake of understandability, we will show the mockups and diagrams corresponding to the “provide books by category” task. As shown in Fig. 13 the mockup for this task adds a combo-box in the home page that the user can change to filter the books according to the selected category. Also, the mockup shows its corresponding title and how many books have been found.

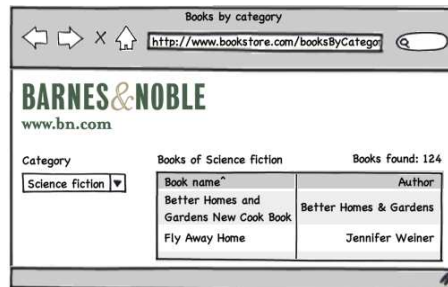


Fig. 13. “Provide books by category” mockup

In Fig. 14 we show a WebSpec diagram that specifies this scenario. Basically, the user starts being located in the Home page and can choose a category from a list of categories. After the user selects the category it should navigate to a different page that contains the title and a list of items. The invariant of this interaction is as follows: $\text{CategoryResult.title} = \text{"Books of " + \{category\}} \ \&\& \ \text{CategoryResult.bookSize} > 0$. The invariant states that the title should be valid according to the selection we have done on the previous combo-box and that there should be at least 1 product in the list.

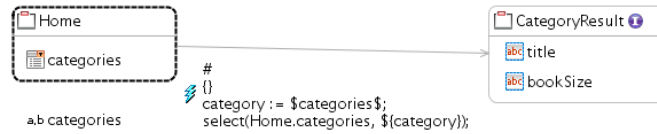


Fig. 14. “Provide books by category” WebSpec diagram

5.2 Generation of Domain and Navigational Models

In Fig. 15 we can see the Domain model which has been derived from the specified requirements. As explained in Sect. 3, to derive the Domain model we take into account the content and service requirements as well as the existence of service or navigational patterns. In this case we can see that five domain classes are created by applying the *Content2DomainClass* transformation rule: one class is generated for each content requirement specified in the SR model. Moreover, we detect three service patterns (see Fig. 9(b)), so operations are added to the classes *client*, *cart* and *book* by executing the *Service2Operation* rule. Finally we detect that the *Provide Book Info* requirement follows the navigational pattern as we can see in Fig. 9(a). In this case the rule *Navigation2Relationship* adds associations among all the resources found in this pattern. The generated Domain model is shown in Fig. 15.

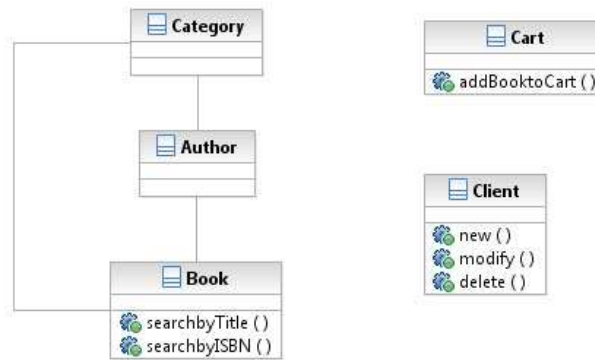


Fig. 15. Generating a Domain model

In the case of the Navigational model, the rule *Nav&Pers2NavClass* is performed adding a home page with a collection of links (i.e. menu). Afterwards, one NC is created for each nav-

igational and personalization requirement with an attached resource, in this case we have five NC created from navigational and personalization requirements. From the menu, a transversal link to each of the created NCs is added (L1 to L4).

The next step consists in checking the navigational and service patterns. In this example, we find a navigational pattern (see Fig.9(a)) where we apply the *Navigation2TLink* transformation creating a transversal link from the NCs created by the associated navigational requirements, to the NC that is represented by the root navigational requirement. In this case two links are added: L5 and L6.

Finally, as we are referring to the website stakeholder, we find three service patterns from which the operations of the NCs books and cart are added and the service links L7, L8 and L9 are created with an associated target NC by applying the *Service2Service&SLink*.

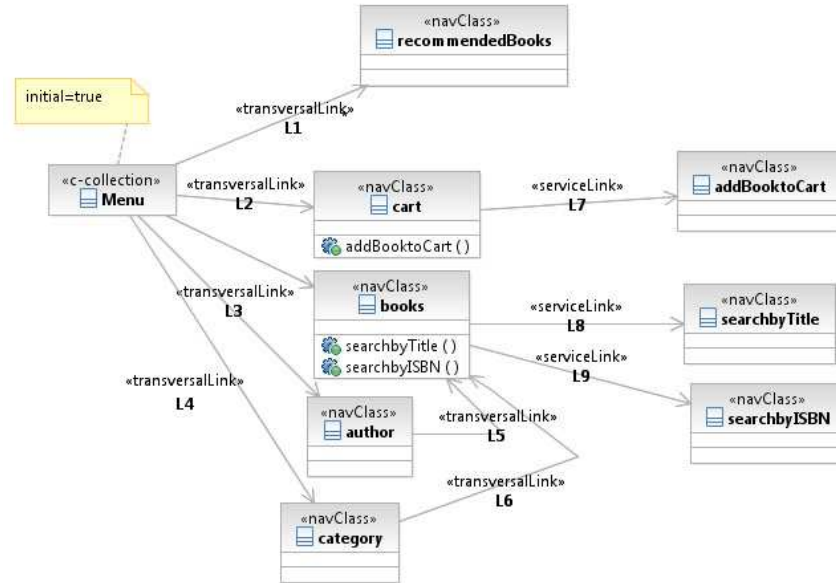


Fig. 16. Generating a Navigation model

5.3 Goal Evaluation

After we obtain and refine the models for our example, our evaluation algorithm should be applied. For our example, we have set $X = 80\%$, $Y = 60\%$ and $Z = 40\%$.

In our sample scenario up to 9 tests of 10 of a WebSpec diagram associated with the “provide books by title”, “provide books by category”, and “provide books by author” tasks are satisfied and only 1 test of 4 of a WebSpec diagram associated with the “provide recommended books” task is satisfied. Following the previously defined policy, the “provide books by title” task is satisfied and the “provide recommended books” is not satisfied (25% of the test passed). A sample of the *i** model already evaluated with a starting configuration of labels from our test cases are shown in Fig. 17 (starting labels are given to the tasks in grey). To sum up, if we implement the Web application by taking into account these test cases, then we will obtain an application that achieves the softgoal “easy to maintain” but neglects the

main goal “Book sales be managed” (deduced by applying the algorithm in [25]).

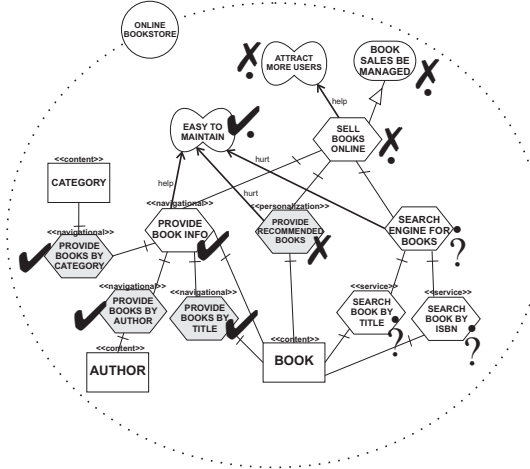


Fig. 17. The result of evaluating the i* model after running the tests

5.4 Implementation Framework

The presented approach has been implemented by using the *Eclipse development platform*⁹. *Eclipse* is a framework which can be extended by means of plugins in order to add more features and new functionalities. A plugin that supports both defined UML profiles for i* has been developed. This new plugin implements several graphical and textual editors. Fig. 18 shows an overview of the tool: the palette for drawing the different elements of i* can be seen on the right-hand side of the figure, while a sample SR model is shown in the center of the figure. Generation rules are also being defined and tested in our prototype.

Implementation of our Web requirements i* model. Our implementation of the i* framework for Web requirements consists of a UML profile which incorporates a number of taxonomic features that enable Web requirements specification. With the implementation of this UML profile has been possible to implement the i* framework in Web to model the needs and expectations of the stakeholders of the Web application. The special features incorporated into the i* framework have allowed that elements of the model can be stereotyped using the requirements taxonomy presented in Sect. 2.

Implementation of A-OOH domain model. The domain model in A-OOH is represented by an UML class diagram, for this reason we have implemented the UML 2.0 meta-model using the Eclipse facilities to represent only the elements necessary to establish a UML class diagram.

Implementation of A-OOH navigational model. The A-OOH navigational metamodel represents the key to the derivation of the navigational model. The implementation was developed using UML profiles.

⁹<http://www.eclipse.org>

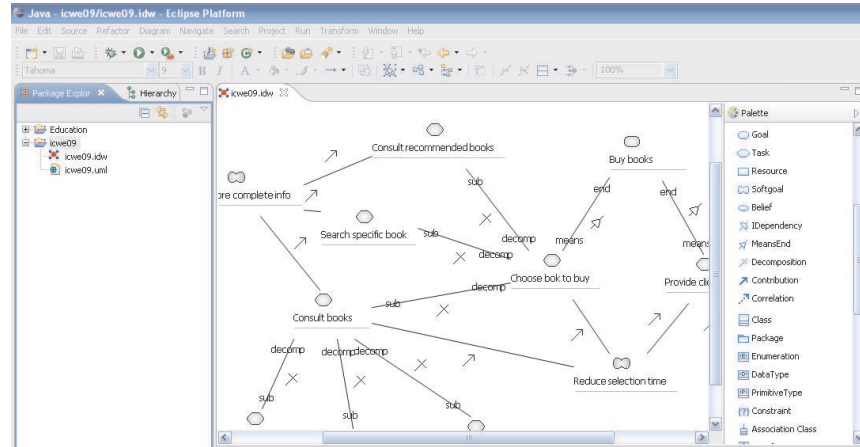


Fig. 18. Screenshot of our prototype

Implementation of the QVT transformation rules. Throughout the paper, QVT has been used as a language for formalizing transformations between models, thus ameliorating the understandability of the transformation process. However, once the transformations have been modeled, they have to be implemented. To this aim, the QVT transformation rules presented above has been implemented using the mediniQVT transformation engine.

Integration with WebSpec. The i* plugin can be easily used together with the WebSpec's Eclipse plugin, thus allowing us to seamless integrate the i* model and the WebSpec diagrams. In Fig. 19 a screenshot of this plugin is shown.

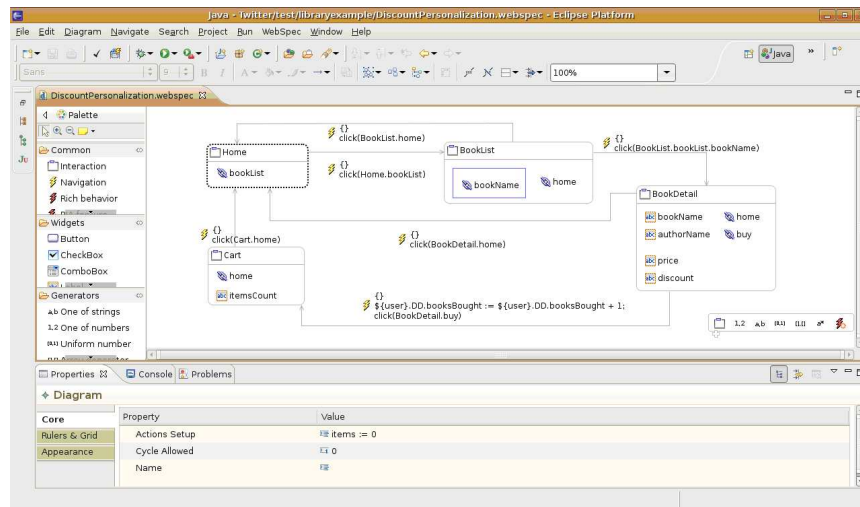


Fig. 19. WebSpec's Eclipse plugin

6 Related Work

Few approaches have focused on defining an explicit requirement analysis stage to model user needs. We can stress the following:

NDT [13] considers a complete taxonomy for the specification of Web requirements. It allows to specify requirements by means of use cases diagrams and templates. It uses a different template for each requirement type they consider, so requirements and objectives are described in a structured way. UWE [7] also describes a taxonomy for requirements related to the Web. It proposes extended use cases, scenarios and glossaries for specifying requirements. WebML [5] also proposes the use of use case diagrams combined with activity diagrams and semi-structured textual description. WSDM [3] is an audience driven approach in which they do a classification of the requirements and the audience. These classes are represented with a diagram in which they are related. Then they are modeled into detail in a Task model using concurrent task trees. OOHDM [26] captures the requirements in use case diagrams. They propose the use of UIDs (user interaction diagrams) for defining the requirements related to navigation which are derived from the Use cases. OOWS [27] focuses on the specification of tasks. They extend the activity diagrams with the concept of interaction point to describe the interaction of the user with the system.

Furthermore, generation of conceptual models from the requirements is an important issue to bridge the gap between users' needs and Web design. To the best of our knowledge, there are two approaches that support this in some way: OOWS provides automatic generation of (only) navigation models from the tasks description by means of graph transformation rules, while NDT [28] defines a requirement metamodel and allows to transform the requirements model into a content and a navigational model by means of QVT rules. Our approach for deriving conceptual models resembles NDT since we have also adopted QVT in order to obtain design artifacts from Web requirements, but we have kept the benefits of the i^* framework by means of the defined profiles and patterns.

However, some of these approaches present the following drawbacks: (i) they do not take into consideration a complete taxonomy of requirements which is suitable in Web applications, or (ii) they consider non-functional requirements in an isolated manner, or (iii) they mainly focus on design aspects of the intended Web system without paying enough attention to Web requirements. Furthermore, none of them perform the analysis of the users' needs. Requirements are figured out by the designer, it may be needed to re-design the website after doing usability and satisfaction tests to the users. Modeling users allow us ensuring that the Web application satisfies real user needs and goals and the user is not overwhelmed with functionalities that he does not need or expect and he does not miss functionalities that were not implemented.

To the best of our knowledge, the only approaches that use goal oriented techniques have been presented in [29, 30]. They propose a complete taxonomy of requirements for the Web and use the i^* notation to represent them. Unfortunately, they do not benefit from every i^* feature, since they only use a metamodel that has some of its concepts, e.g. means-end, decomposition or contribution links from i^* are not specified in the approach presented in [29]. Our approach not only benefits from i^* features but also used with mockups and WebSpec diagrams can provide automatic evaluation of its models. This feature is extremely important to get feedback during the development process of a Web application.

On the other hand, in a goal-oriented requirement engineering approach, goal evaluation is important to check whether the goals and needs of the stakeholders are satisfied. Specially in the Web engineering field, where the continuous participation of the stakeholders during the process is vital to obtain feedback. There are some approaches that have goal evaluation such as the NFR Framework [31]. In this framework qualitative labels are propagated throughout a Softgoal Interdependency Graph (SIG), and similar to the goal evaluation procedure of [25], the user must resolve the conflicts [31]. In [32] there are some guidelines on how to extend this procedure to be used with i*. Despite this procedure could be applied to i* models, it should be adapted to be used to provide feedback to stakeholders. The resolution of conflicts and the integration with late requirement analysis artifacts like WebSpec are the main drawbacks we have found. Our approach integrates seamlessly with a detail requirement analysis and provides an automatic way of evaluating if the goals are been satisfied by the application under development.

GRL [33], a variant of i*, has a fully automated evaluation method but does not allow to make decisions in the presence of conflicting, partial or unknown information. The hard-coded rules used to resolve softgoals often result in the proliferation of conflicts or partial values. Moreover, this approach should be adapted to Web engineering to provide automatic generation of the models and automatic requirements validation as shown in this paper.

7 Conclusions and Future Work

Websites require special techniques for requirement analysis in order to reflect, from early stages of the development, specific needs, goals, interests and preferences of each user or user type. However, Web engineering field does not pay the attention needed to this issue. We have presented a goal oriented approach on the basis of the i* framework to specify Web requirements. It allows the designer to make decisions from the very beginning of the development phase that would affect the structure of the envisioned website in order to satisfy users.

We have improved the requirements phase by complementing i* models with mockups and WebSpec diagrams to provide a more detailed analysis of interactive requirements. Also, a first version of the domain and navigational models are obtained from the i* model allowing developers to have a starting point for model refinement. During the refinement process, users can observe and provide feedback of the progress by looking at the automatic evaluation of the i* model. This evaluation is performed by executing the automatic derived tests, generated from WebSpec, against the application under development.

Our short-term future work consists of completing the transformation rules in order to obtain the rest of the A-OOH models (i.e. presentation and personalization models). Finally, as long-term future work we plan to carry out a set of experiments to measure the effectiveness of our proposal.

Acknowledgements

This work has been partially supported by the MESOLAP project (TIN 2010-14860) from the Spanish Ministry of Education and Science, by the QUASIMODO project (PAC08-0157-0668) from the Castilla-La Mancha Ministry of Education and Science (Spain), and by the MANTRA project (GRE09-17) from the University of Alicante (Spain).

References

1. S. Casteleyn, W. V. Woensel, and G.-J. Houben. A semantics-based aspect-oriented approach to adaptation in Web engineering. In *Hypertext*, pages 189–198, 2007.
2. C. Cachero and J. Gómez. Advanced conceptual modeling of Web applications: Embedding operation interfaces in navigation design. In *JISBD*, pages 235–248, 2002.
3. S. Casteleyn, I. Garrigós, and O. D. Troyer. Automatic runtime validation and correction of the navigational design of Web sites. In *APWeb*, pages 453–463, 2005.
4. N. Koch. Software engineering for adaptive hypermedia systems: Reference model, modeling techniques and development process. *Softwaretechnik- Trends*, 21(1), 2001.
5. S. Ceri and I. Manolescu. Constructing and integrating data-centric web applications: Methods, tools, and techniques. In *VLDB*, page 1151, 2003.
6. G. Rossi, D. Schwabe, and R. Guimarães. Designing personalized Web applications. In *WWW*, pages 275–284, 2001.
7. N. Koch. Reference model, modeling techniques and development process software engineering for adaptive hypermedia systems. *KI*, 16(3):40–41, 2002.
8. I. Garrigós. *A-OOH: Extending Web Application Design with Dynamic Personalization*. PhD thesis, University of Alicante, Spain, 2008.
9. F. Daniel, M. Matera, A. Morandi, M. Mortari, and G. Pozzi. Active rules for runtime adaptivity management. In *AEWSE*, 2007.
10. R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
11. E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Canada, 1995.
12. E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE*, pages 226–235, 1997.
13. M. J. Escalona and N. Koch. Requirements engineering for Web applications - a comparative study. *J. Web Eng.*, 2(3):193–212, 2004.
14. D. C. Nguyen, A. Perini, and P. Tonella. A goal-oriented software testing methodology. In *AOSE*, pages 58–72, 2007.
15. E. Robles, I. Garrigós, J. Grigera, and M. Winckler. Capture and evolution of Web requirements using WebSpec. In B. Benatallah, F. Casati, G. Kappel, and G. Rossi, editors, *ICWE*, volume 6189 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2010.
16. QVT Language. <http://www.omg.org/cgi-bin/doc?ptc/2005-11-01>.
17. I. Garrigós, J.-N. Mazón, and J. Trujillo. A requirement analysis approach for using i* in Web engineering. In *ICWE*, pages 151–165, 2009.
18. H. Estrada, A. M. Rebollar, O. Pastor, and J. Mylopoulos. An empirical evaluation of the i* framework in a model-based software generation environment. In *CAiSE*, pages 513–527, 2006.
19. M. Strohmaier, J. Horkoff, E. S. K. Yu, J. Aranda, and S. M. Easterbrook. Can patterns improve i* modeling? two exploratory studies. In *REFSQ*, pages 153–167, 2008.
20. A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Practice and Promise of The Model Driven Architecture*. Addison Wesley, 2003.
21. K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Technique in the Context of the Model Driven Architecture*, Anaheim, October 2003.
22. A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2002.
23. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.
24. OCL. <http://www.omg.org/cgi-bin/doc?ptc/03-10-14>.
25. J. Horkoff and E. Yu. Evaluating goal achievement in enterprise modeling an interactive procedure

- and experiences. In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, A. Persson, and J. Stirna, editors, *The Practice of Enterprise Modeling*, volume 39 of *Lecture Notes in Business Information Processing*, pages 145–160. Springer Berlin Heidelberg, 2009. 10.1007/978-3-642-05352-812.
26. D. Schwabe and G. Rossi. An object oriented approach to Web-based applications design. *TAPOS*, 4(4):207–225, 1998.
 27. P. Valderas, V. Pelechano, and O. Pastor. A transformational approach to produce Web application prototypes from a web requirements model. *Int. J. Web Eng. Technol.*, 3(1):4–42, 2007.
 28. N. Koch, G. Zhang, and M. J. Escalona. Model transformations from requirements to Web system design. In *ICWE*, pages 281–288, 2006.
 29. D. Bolchini and P. Paolini. Goal-driven requirements analysis for hypermedia-intensive Web applications. *Requir. Eng.*, 9(2):85–103, 2004.
 30. F. M. Molina, J. Pardillo, and J. A. Toval. Modelling Web-based systems requirements using WRM. In *WISE Workshops*, pages 122–131, 2008.
 31. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering (THE KLUWER INTERNATIONAL SERIES IN SOFTWARE ENGINEERING Volume 5)*. Springer, 1st edition, October 1999.
 32. L. Liu and E. Yu. Designing information systems in social context: a goal and scenario modelling approach. *Inf. Syst.*, 29(2):187–203, 2004.
 33. D. Amyot, S. Ghanavati, J. Horkoff, G. Mussbacher, L. Peyton, and E. Yu. Evaluating goal models within the goal-oriented requirement language. *Int. J. Intell. Syst.*, 25(8).
 34. i* wiki. <http://istar.rwth-aachen.de>.

Specifying personalizable and accessible web applications with WebSpec

The content of this chapter corresponds with the following papers:

*Medina, N. M., Burella, J., Rossi G., Grigera J., **Robles Luna E.** An Incremental Approach for Building Accessible and Usable Web Applications. Proceedings of the 11th International Conference on Web Information System Engineering (**WISE 2010**). Hong Kong, China. Acceptance rate: 18.8%. Core A.*

***Robles Luna E.**, Garrigos I., Rossi G. Capturing and Validating Personalization Requirements in Web Applications. Proceedings of the 1st Workshop on The Web and Requirements Engineering (**WeRE 2010**). Sydney, Australia.*

In the previous papers we have shown how to use WebSpec not only in the context of WebTDD but also in conjunction with early requirements in i*. However, we have only concentrated in functional requirements (those who affect the functionality of the web application).

In this chapter we show how to use WebSpec for the specification of non functional requirements like accessibility and personalization of Web applications. In each case we provide small extensions to the core language with the intent of allowing the specification of these requirements in the context of WebTDD.

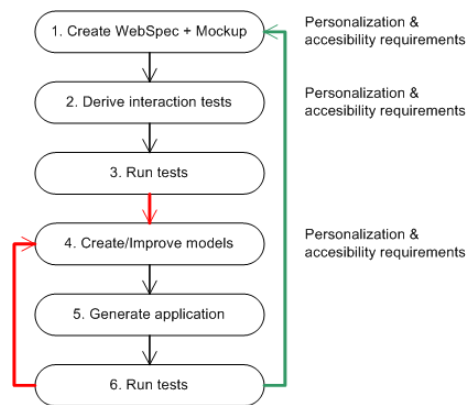


Fig. 6.1. Specifying personalizable and accessible applications with WebSpec in WebTDD

The content of this chapter are two papers published in the *International Conference on Web Information System Engineering (WISE)* and in the *International Workshop on the Web and Requirements Engineering (WeRE)*. The aim of WISE is to provide an international forum for researchers, professionals, and industrial practitioners to share their knowledge in the rapidly growing area of Web technologies, methodologies and applications. On the other hand, the aim of WeRE is to be an international forum for exchanging ideas on both using Web technologies as a platform for requirements engineering, and applying requirements engineering in the development and use of web-based applications.

An Incremental Approach for Building Accessible and Usable Web Applications

Nuria Medina Medina¹, Juan Burella^{2,4}, Gustavo Rossi^{3,4}, Julián Grigera³,
Esteban Robles Luna³

¹Departamento de Lenguajes y Sistemas Informáticos, Universidad de Granada, España
nmedina@ugr.es

²Departamento de Computación, Universidad de Buenos Aires, Argentina
jburella@dc.uba.ar

³LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
{gustavo, julian.grigera, esteban.robles}@lifia.info.unlp.edu.ar

⁴Also at CONICET

Abstract. Building accessible Web applications is difficult, moreover considering the fact that they are constantly evolving. To make matters more critical, an application which conforms to the well-known W3C accessibility standards is not necessarily usable for handicapped persons. In fact, the user experience, when accessing a complex Web application, using for example screen readers, tends to be far from friendly. In this paper we present an approach to safely transform Web applications into usable and accessible ones. The approach is based on an adaptation of the well-known software refactoring technique. We show how to apply accessibility refactorings to improve usability in accessible applications, and how to make the process of obtaining this “new” application cost-effective, by adapting an agile development process.

Keywords: Accessibility, Visually Impaired, Web engineering, TDD, Web requirements.

1 Introduction

Building usable Web applications is difficult, particularly if they are meant for users with physical, visual, auditory, or cognitive disabilities. For these disadvantaged users, usability often seems an overly ambitious quality attribute, and efforts in the scientific community have been generally limited to ensure accessibility. We think accessibility is a good first step, but not the end of the road. Usability and accessibility should go hand in hand, so disabled users can access information in a usable way, since it is not fair to pursue usability for regular users and settle with accessibility for disabled users. Thus we consider that the term Web accessibility falls short and should be replaced by the term “usable web accessibility” or “universal usability”, whose definition can be obtained from the combination of the two quality attributes. As an example, let us suppose a blind person accessing a (simplified) Web application like the one shown in Figure 1, using a screen reader [1]. To enforce our

statement, we assume that this application fulfils the maximum level of accessibility, AAA, according to the Web Content Accessibility Guidelines (WCAG) [2]. This means that the HTML source of the application satisfies all the verification points, which check that all the information is accessible despite any user's disabilities. However, using the screen reader, it will be difficult for the blind user to go directly to the central area of the page where the books' information is placed. On the contrary, he will be forced to listen (or jump) one by one all the links before that information can be listened (even when he does not want to use them).

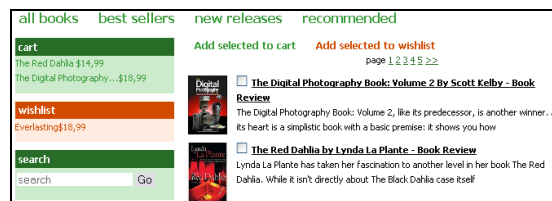


Fig. 1. Accessible but not usable page

The main problem, that we will elaborate later, is that this page has been designed to be usable by sighted users and “only” accessible by blind users. In this paper we present an approach to systematically and safely transform an accessible web application into a usable and accessible (UA) one. The approach consists in applying a set of atomic transformations, which we call accessibility refactorings, to the navigational and interface structure of the accessible application. With these transformations we obtain a new application that can be accessed in a more friendly way when using, for example, a screen reader. We show, in the context of an agile approach, how this strategy can be made cost feasible, particularly when the application evolves, e.g. when there are new requirements. Our ideas are presented in the context of the WebTDD development approach [3], but they can be applied either in model-driven or coding-based approaches without much changes. Also, while the accessibility refactorings we describe are focused on people with sight problems, the approach can be used to improve usability for any kind of disability.

The main contributions of the paper are the following: a) we introduce the concept of usable accessibility b) we show a way to obtain a UA web application by applying small, behaviour-preserving, transformations to its navigation and interface structures; c) we demonstrate the feasibility of the approach by showing not only how to generate the UA application but also how to reduce efforts when the application evolves. The rest of the paper is structured as follows: Section 2 discusses some related work in building accessible applications; section 3 presents the concept of accessibility refactoring and briefly outlines some refactorings from our catalogue. In section 4 we present the core of the approach using the example of Figure 1, and finally section 5 concludes the paper and discusses some further work we are pursuing.

2 Related Work

Web usability for visually impaired users is a problem that is far from being solved. The starting point of the proposed solutions leans on two basic supports: the WCAG Guidelines [2] and the screen readers [1] used together with traditional or “talking” browsers [4]. Then, the methods and proposed tools to achieve accessibility and usability in the Web diverge in two directions [5]: assessment or transformation. In the first group, the automated evaluation tools, such as Bobby (Watchfire) [6], analyze the HTML code to ensure that it conforms to accessibility or usability guidelines. In the second group, automated transformation tools help end users, rather than Web application developers. These tools dynamically modify web pages to better meet accessibility guidelines or the specific needs of the users.

Automated transformation tools are usually supported by some middleware, and they act as an intermediary between the Web page stored in the server, and the Web page shown in the client. Thus, in the middleware, diverse transcodings are performed. An example is the middleware presented in [7], which is able to adapt the Web content on-the-fly, applying a transcoding to expand the context of a link (the context is inferred from the text surrounding the link), and other transcoding to expand the preview of the link (processing the destination of the link). Other example is the proposal in [8], in which semantic information is automatically determined from the HTML structure. Using these semantics, the tool is able to identify blocks and reorganize the page (grouping similar blocks, i.e. all the menus, all the content areas, etc). This will create sections within the page that allow users to know the structure of the page and move easily between sections (ignoring non essential information for him). However, none of these automatic transcodings are enough to properly reduce the overhead of textual and graphic elements, as well as links, which clutter most pages (making their reading through a screen reader very noisy). This is because discerning meaningful from accessory content is a task that must be manually performed. A basic example of “manual” transcoding is the accessible method proposed in [9], which uses stylesheets to hide text (marked with a special label) from the page prepared for sighted users. Another interesting example is Dante [10], a semi-automated tool capable of analyzing Web pages to extract objects which are meaningful for the handicapped person during navigation, discover their roles, annotate them and transform pages based on the annotations. In [11] meanwhile, Dante annotations are automatically generated in the design process. In this case, the intervention of the designer is performed in the phase of modeling, but still needed.

We believe that the problem of usability for impaired people must be attacked from the early stages of applications design. Furthermore, all stakeholders (customers, designers and users) must be involved in the process. Hence, instead of proposing an automatic transcoding tool, we provide a catalogue of refactorings that the designer can apply during the development process, and later during the evolution of the Web application. The catalogue is independent of the underlying methodology and development environment, so refactorings can be integrated into traditional life cycle models or agile methodologies. However, to emphasize our point we show how a wise combination of agile and model-driven approaches can improve the process and allow the generation of two different applications, one for “normal” users and another, which provides usable accessibility for impaired users.

3 Making Accessible Web Applications More Usable

Achieving universal usability is a gradual and interdisciplinary process in which we should involve all application's stakeholders. In addition, we think that it is a user-centred process that must be considered in early phases of the design of Web applications. For the sake of conciseness, however, we will stress out the techniques we use, more than the process issues, which will be briefly commented in Section 4.

The key concept in our approach is refactoring for accessibility. Refactoring [12] was originally conceived as a technique to improve the design of object-oriented programs and models by applying small, behaviour-preserving, transformations to the code base, to obtain a more modular program. In [3] we extended the idea for Web applications with some slight differences with respect to the original approach: the transformations are applied to the navigational or presentation structures, and with the aim of improving usability rather than modularity. In this context we defined an initial catalogue of refactorings, which must be applied when a bad usability smell [13] is detected. More recently, in [14] we extended the catalogue incorporating a new intent: usable accessibility. As said before, we will concentrate on those refactorings targeted to sight disabled persons. Subsequently, section 3.1 briefly describes the specific catalogue of refactorings to achieve UA for sight impaired users.

3.1 The Refactoring Catalogue

Each refactoring in our catalogue to improve UA, specifies a concrete and practical solution to improve the usability of a Web application, that will be accessed by a visually impaired user. Each UA refactoring is uniformly specified with a standard template, so it can be an effective means of communication between designer and developers. The basic points included in the template are three: purpose, bad smells and mechanics. The purpose, defined in terms of objectives and goals, establishes the property of usability to be achieved with the application of the refactoring. The bad smells are sample scenarios in which it is appropriate to apply the refactoring, that is, elements or features of the Web site which generate a usability problem. Finally, each mechanics explains, step by step, the transformation process needed to apply the refactoring and thus solve the existing usability problem.

The refactorings included in the catalogue are divided in two groups: Navigation Refactorings and Presentation Refactorings. Navigation refactorings try to solve usability problems related to the navigational structure of the Web application. Therefore, the changes proposed by this first type of refactorings modify the nodes and links of the application. Presentation refactorings meanwhile propose solutions to usability problems whose origin are the pages' interfaces. Therefore this second type of refactorings implies changes in the appearance of the Web pages.

Concretely, the navigation refactorings included in the UA catalogue allow: to split a complex node, to join two small nodes whose contents are deeply related, to make easier the access between nodes creating new links between them, to remove an unnecessary link, information or functionality with the aim to simplify the node without losing significant content or, conversely, to repeat a link, functionality or information contained on a node in another node where it is also necessary and its inclusion does not overload the resulting node, etc.

Presentation refactorings included in the UA catalogue determine when and how: divide a complex and heterogeneous page in a structure of simpler pages, combine two atomic pages in a cohesive page, add needed anchors, remove superfluous anchors, add contextual information such as size indicators in dynamic list and tables, distribute or duplicate the options of a general menu for each one of the items valid for the menu, replace pictures and graphics for an equivalent specific text or remove the figure if it is purely aesthetic, reorder the information and functionality on the page in a coherent order to read and use, reorganize panels and sections to be read from top to bottom and from left to right, fix the floating elements, transform nested menus into linear tables more easier to read, etc.

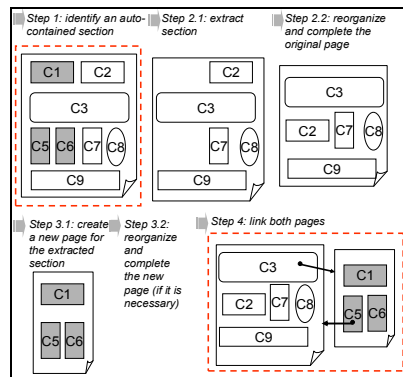


Fig. 2a. “Split Page”

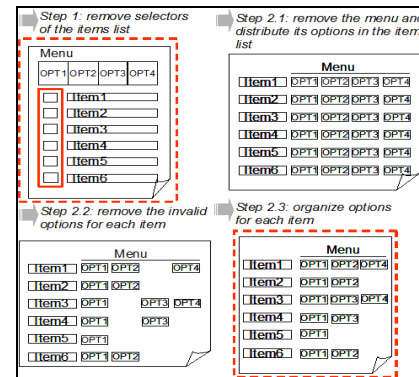


Fig. 2b. “Distribute General Menu”

Figure 2a shows the steps needed to put into operation the “Split Page” presentation refactoring. As shown in the figure, the application of this refactoring involves simplifying an existing page, identifying and extracting self-contained blocks of information and functionality (steps 1 and 2.1), and then, creating one or more pages with the information / functionality extracted from the original page (step 3.1). Both, the original page (step 2.2) as the news pages (step 3.2) must be structured (that is, to organize the information for their appropriate reading and viewing), and can be supplemented or not with other additional information. Finally, the original page and the new pages must be linked together (step 4). Most refactorings allow several alternatives for certain steps in their mechanics. For the sake of conciseness Figure 2a shows the “normal” course of the “Split Page” refactoring. In Section 4 we illustrate the use of this refactoring in a concrete example.

Figure 2b shows the mechanism of the refactoring “Distribute General Menu”, which proposes to remove the general menu affecting a list of elements by adding the menu actions to each element. The selectors of elements (e.g. checkboxes) are also removed in the container as the operations are now locally applied to each element.

In most cases, when solving a usability problem we need to update both navigation and presentation levels. Thus, many navigation refactorings have associated an automatic mechanism for changes propagation, which implies the execution of one or more presentation refactorings. More details can be read at [14]. We next show how we use the ideas behind accessibility refactorings in an agile development process.

4 Our Approach in a Nutshell

Along this section, we will show how we use the catalogue of accessibility refactorings to make the development of UA Web Applications easier. In a coarse grained description of our approach, we can say that it has roughly the same steps that any refactoring-based development process has (e.g. see [12]), namely: (a) capture application requirements, (b) develop the application according to the WACG accessibility guidelines, (c) detect bad smells (in this case UA bad smells), and (d) refactor the application to obtain an application that does not smell that way, i.e. which is more usable, besides being accessible.

Notice that step *b* (application development) may be performed in a model-based way, i.e. creating models and deriving the application, or in a code-based fashion, therefore developing the application by “just” programming. Step *c* (detecting bad smells) may be done “manually”, either by inspecting the application, by performing usability tests with users, or by using automated tools. Finally, step *d*, when refactorings are applied, may be manually performed following the corresponding mechanics (See Figure 2), or automatically performed by means of transformations upon the models or the programming modules. A relevant difference with regard to the general process proposed in [12] is that in our approach step *d* is only applied when the application is in a stable step (e.g. a new release is going to be published) and not each time we add a new requirement. Anyway, for each accessibility refactoring we perform a short cycle, to improve the application incrementally.

One important concern that might arise regarding this process is that it might be costly, particularly during evolution. Therefore, we have developed an agile and flexible development process, and a set of associated tools which guarantee that we can handle evolution in a cost-effective way [15]. For the sake of conciseness, we will focus only on the features related to accessibility rather than evolution issues which are outside the scope of the paper. We discuss them as part of our further work.

Concretely, we use WebTDD [3], an agile method that puts much emphasis in the continuous involvement of customers, and comprises short development cycles in which stakeholders agree on the current application state. WebTDD uses specific artefacts to represent navigation and interaction requirements, which we consider to be essential for accessible applications. Similarly to Test-Driven Development (TDD) [16], WebTDD uses tests created before the application is developed to “drive” the development process. These tests are used later to verify that requirements have been corrected fulfilled. Different from “conventional” TDD, we complement unit tests with interaction and navigation tests using tools like Selenium (<http://seleniumhq.org/>). Figure 3 shows a simplified sketch of the development process. In the first step (1) we “pick” a requirement (e.g. represented with use cases or user stories) and in (2) we agree on the look and feel of the application using mockups. We capture navigation and interaction requirements, and represent them using WebSpec [17], a domain-specific language (DSL) which allows automatic test generation and tracking of requirement changes. At this point we can exercise mockups and simulate the application, either using a browser or a screen reader; therefore we can check accessibility guidelines and have early information on the need to refactor to improve usable accessibility in the step 7. Next (3), we derive the interaction tests from the WebSpec diagrams, and run them (4); it’s likely that these

tests will fail, indicating the starting point to begin the development to make tests pass. As said before, step 5 might imply dealing with models (generating code automatically), coding or a combination of both. In step 6, we run the tests again and iterate the process until all tests pass. Once we have the current version of the application ready we repeat the cycle with a new requirement (steps 1 to 6).

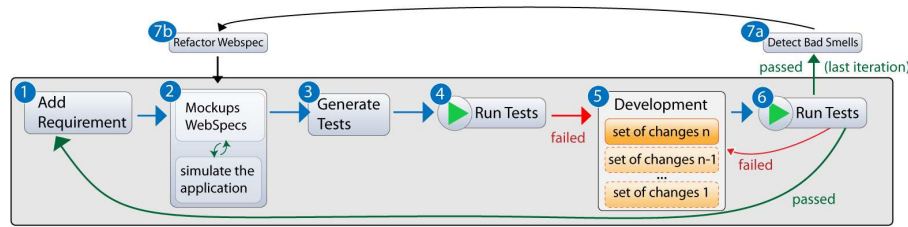


Fig. 3. WebTDD process for UA applications.

After we reach a stable state of the application and we want to publish its current version, we look for bad accessibility smells and identify the need for UA refactorings (7.a). Next, we perform short cycles by applying each refactoring to the WebSpec diagrams containing the detected bad smells (7.b). The altered WebSpecs will generate new tests that check the new accessibility/usability features, and propagate the changes to the code (or model) to obtain the new UA application (steps 2 to 6).

In the next subsections we explain some of these aspects in a more detailed way focusing on UA development. To illustrate our approach, we will use the development of a simplified online book store (as the one shown in Figure 1), and when possible, we ignore the activities related with tests since they are outside the scope of the paper.

4.1. Gathering Navigation and Interface Requirements

Navigation and interface requirements are captured early in the development cycle through mockups and WebSpecs (step 2 in Figure 3). User interface Mockups help to establish the look and feel of the applications, along with other broad interaction aspects. They can be elaborated using plain HTML or commercial tools such as Balsamiq (<http://www.balsamiq.com>). Mockups can be easily adjusted to comply with accessibility guidelines. Figure 1 showed a mockup for our example's homepage.

WebSpecs are simple state machines that represent interactions as states and navigations as transitions, adding the formal power of preconditions and invariants to assert properties in states. An “interaction” represents a point where the user consumes information (expressed with interface widgets), and interacts with the application by using some of its widgets. Some actions (clicking a button, adding some text in a text field, etc) might produce “navigation” from one “interaction” to another and, as a consequence, the user moves through the application's navigation space.

Figure 4 shows a simplified WebSpec diagram that specifies the navigations paths from the BookList interaction and is related with the mockup of Figure 1. In the

BookList “interaction” the user can authenticate, add books to the cart or to the wish list and search books. This diagram is the starting point for developing our simplified book store application, as it has key information to specify (at least partial) navigational models (as shown in [3]). Additionally, WebSpecs allow the automatic generation of navigation tests for the piece of functionality it represents, and with the aid of a tool suite, it records requirements changes, to trace and simplify implementation changes. In this sense, every changes made in a WebSpec (even the “initial” WebSpec as a whole) are recorded as “first class” change objects (as shown in Figure 4); these objects are later related with the corresponding model (or implementation) artefacts to improve traceability and automatic change management, using effect managers as explained in [15]. Specifically, each feature of the WebSpec in Figure 4 is traced to the corresponding modelling elements; in this way, when some of these features change, there is a way to automatically (or with minor designer intervention) change the corresponding models or programs. Therefore, step 5 in Figure 3 is viewed as the incremental application of these changes to the current implementation.

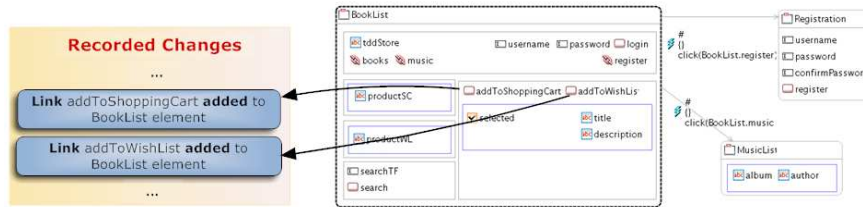


Fig. 4. *BookList* interaction in the *Books Store* WebSpec

4.2 Deriving an Accessible Application

In our approach we do not prescribe any particular development style, though we have experienced with model-driven (specifically, WebML [18]) and code-based (with Seaside- www.seaside.st/) approaches. Once we run the tests for a requirement and noticed that they fail, we build the corresponding models to satisfy such tests and derive the application (steps 3 to 6, in Figure 3). The construction of these models is an incremental task, managing the effects of each recorded change (step 5 in Figure 3). In addition to the changes log, we record the relationship between the WebSpec elements and its counterpart in the model, necessary for the automation of future changes on these elements. This recording is done at this stage, when change effects are managed and the model is built. For example, when the ‘add to cart’ addition link is managed, a counterpart element is added to the model and the relationship between both elements is recorded. Then, if we need to manage a change to configure any property of this element (e.g. its value), this can be automated since the change management tool knows its representation in the model.

In this stage, accessibility can be addressed using any of the approaches cited in Section 2, for example by incorporating Dante annotations in the corresponding model-driven approach (See [10, 11]). Alternatively, we can “manually” work on the

resulting application by improving the HTML pages to make them fully accessible. In both cases, given the nature of the WebTDD approach, the improvement is incremental; in each cycle we produce an accessible version of the application.

In traditional Web application development, accessibility is tackled as a monolithic requirement that must be satisfied by the application which is checked by running accessibility tests such as TAW (<http://www.tawdis.net/>). What is different in our approach is that we do not try to make the application accessible in one step; instead, we can decide which tests must be run on each development iteration and specific page. Therefore, in the first iteration we may want to make the “BookList” page accessible and satisfying the accessibility test “Page Titled” (Web pages must have titles that describe topic or purpose) and in the second iteration we may want to do the same with page “Best sellers”. Our approach follows the very nature of agile development trying to incrementally improve the accessibility of an existing application. Stakeholders’ involvement obviously helps in this process. To achieve this goal, we can specify which tests must be run on a specific interaction for each WebSpec diagram. For instance, in the diagram of Figure 4, we can initially run the “Page Titled” test during the first iteration, and the “Headings and Labels” (headings and labels describe topic or purpose) test during the second iteration. This approach helps to improve times during development and allows focusing on a specific accessibility requirement, though we can still execute all accessibility tests for every “interaction” like in traditional Web application development if necessary. From an implementation point of view, this “selective” testing is performed using a Javascript version of the WGAC accessibility tests and executing them depending on the tests selected on the WebSpec diagrams.

4.3 Detecting Bad Accessibility Smells

By following the WebTDD cycle (steps 1 to 6 in Figure 3), we will obtain an accessible application, but not necessarily a UA application. For example, if we analyze the page shown in Figure 1 (accessible according the WCAG), we can see that it presents several bad smells contemplated in the UA catalogue that have been outlined in Section 3.1.

First, the page mixes concepts and functions that are not closely related, such as: shopping cart, wish list, information on books, access to other products and user registration. A sighted user quickly disregards the information in which he is not interested (e.g. the registration if he just wants to take a look) and goes quickly to the area that contains what he wants (e.g. the central area where the available books are listed). However, a blind user does not have the ability to look through; when accessing the page using a screen reader which sequentially reads the page content, he will be forced to listen to a lot of information and functionality in what he may be not interested before reaching the desired content. In order to eliminate this bad smell, the refactoring “Split Page” can be applied. Besides, the actions provided to operate with the products listed in the central area of the page (books in this moment) refer to the selected books in the list; this implies that before applying an action in this menu (for example, add a book to the cart), the book or books must be selected by using checkboxes. This task is trivial for a sighted user, but it is considerably more

complicated for a user who is accessing through a screen reader, as the reader reads the actions first and then the book list. Even though it is possible to scroll through the links on the page with the use of navigation buttons (provided by most readers), moving back (e.g. to look for the option once you have marked the products), can cause confusion and be tedious if the list is long. In order to eliminate this bad smell, the refactoring “Distribute General Menu” can be applied.

Therefore, we conclude that we need to apply some refactorings to obtain a better application. This could be done manually on the final application but it might be difficult to check that we didn’t break any application behaviour. Next, we show how to make this process safer and compatible with the underlying WebTDD process and at the same time settle the basis to simplify evolution.

4.4 Applying Refactorings to the WebSpec Diagrams

As a solution to safely produce UA Web applications from existing ones, we propose to apply accessibility refactorings to the navigation and interaction requirements specifications (step 7 in Figure 3). Since WebSpec is a DSL formally defined in a metamodel [17], these refactorings are essentially model transformations of WebSpec's concepts. Each transformation comprises a sequence of changes on a WebSpec diagram, which are aimed to eliminate a specific bad smell. Moreover, as shown in [15,17] and explained before, these changes are also recorded in change objects that can be used to semi automatically upgrade the application as we will show in Section 4.5.

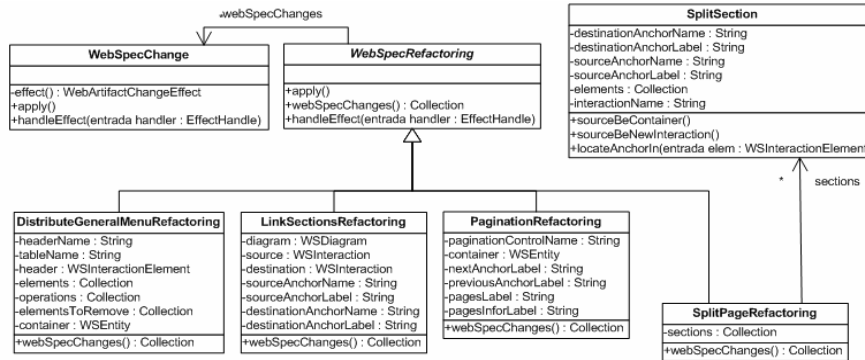


Fig. 5. Refactoring's metamodel.

Usability refactorings are also conceptualized in a metamodel, part of which is shown in Figure 5. Refactorings classes provide an extension to the WebSpec metamodel; this allows grouping a set of changes with a coherent meaning. An interesting point to remark is the fact that these refactorings transform WebSpec diagrams instead of models (or code). This has several advantages; for example, with these diagrams and the corresponding new mockups, we can simulate the application. The new mockups could be automatically generated if they are also “imported” from a metamodel like we do in [19]. Also we automatically generate the new navigation tests to assess if the implementation changes were implemented correctly.

As an example, let us consider the application of the “Distribute General Menu” refactoring to the WebSpec of Figure 4. This refactoring takes as input an item container, elements and menu options to be distributed into this container, and elements to be eliminated for each item. In our example, we configure this refactoring with the “book” element as container, the elements “title” and “description”, the menu options “Add to cart” and “Add to Wish List” and the checkbox to be removed. In Figure 6 we show the result of applying the refactoring, where the “Add to cart” and “Add to Wish List” options are added in each book item (in order to simplify the diagram we only shows the BookList interaction).



Fig. 6. “Distribute Menu” refactoring example.

4.5 Deriving the UA Web Application

Once we applied a refactoring to the needed WebSpec specification, we proceed with the cycle (Figure 3). From now on, we work as we did with “normal” requirements (steps 2 to 6). Once we agreed the new look and feel of the refactored application with the customer (step 2), we generate and run the navigation tests to drive the implementation of these changes (steps 3 and 4). We run these tests; they obviously fail and the process continues with the refactoring effect management (step 5). As we previously explained, refactorings introduce changes in the corresponding WebSpecs, and their explicit representation as first-class objects helps us manage the changes to be applied to the application. As a refactoring is a group of WebSpec changes, we “visit” each of these changes to manage its effects. We start delegating these changes to a Change Management tool, which can automatically (or with some programmer intervention) alter either the models that will in turn generate the new application, or the code in a code-based approach.

In our tool suite we deal with these refactorings in the same way that we manage the changes generated by any new requirement. For example as shown in Figure 7, the “Distribute General Menu” refactoring involves “Move operations” changes; when we manage these changes the “Add to cart” and “Add to Wish List” links are moved into each book item on the application.

We use these changes in the WebSpec specification to improve the development stage, with the aim of reducing the cost of their effects on the application, automating these effects in many cases. Additionally, we are able to determine which tests are affected by each change, to trim the set of required tests that must be performed (see the details of this change management process in [15,17]). Finally, a UA requirement is completed when all tests pass (step 6).

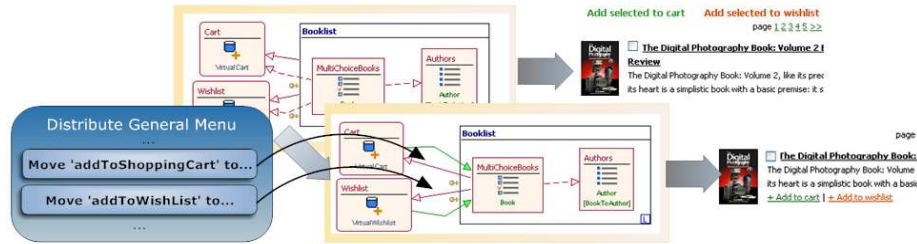


Fig. 7. Handle “Move operations” effects.

In our example, one of the bad smells detected in 4.2 is the way the interaction with the items on the book list is performed: a checklist with general operations to apply on the selected books. In this situation, the refactoring “Distribute General Menu” can be applied in order to improve the usability.

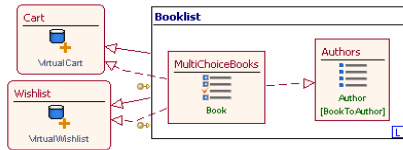


Fig. 8a. General Menu with checklists.

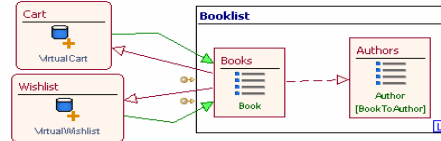


Fig. 8b. Distributed Menu.

Figure 8a shows a WebML diagram for the page that lists all books and lets the user add books to a Shopping Cart or a Wish list. Since the book list is presented as a checkbox set (using a specific WebML unit called “Multi Choice Index Unit”), the user has the ability to check different books and select an action to perform on the selected group, as seen in the units “Cart” and “Whishlist”. The application of this refactoring generates automatically the diagram in Figure 8b, where the book list becomes a simple list (replacing the “Multi Choice Index Unit” unit with a plain “Index Unit”); the actions “Cart” and “Whishlist” are now directly linked from the list, and therefore every item on the Index Unit called “Books” gets individual links to each action. From this new navigational model and the corresponding interface template (derived from the mockup), we are able to derive a UA version of the home page shown in Figure 1. Figure 9a shows the result of the process.



Fig. 9a. Distributed Menu in book list

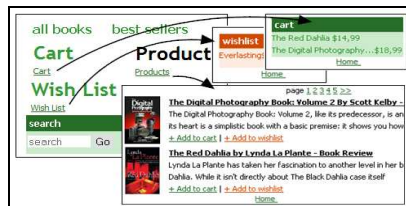


Fig. 9b. A new, usable and accessible home

Another bad smell detected is the mixed up contents on the bookstore’s homepage. To overcome this problem, the “Split Page” refactoring is applied. For the sake of

conciseness we only show the final result of applying the refactoring in the final application. Figure 9b shows the result of the new iteration, where the initial page has been cleaned, extracting in three new pages the information and functionality needed to: list products (BookList page), manage the wish list (WishList page) and manage the shopping cart (ShoppingCart page). After finishing this process we end with two Web applications: the “normal” one and the UA application. From now on, evolution can be tackled in two different ways: by treating the two applications separately or by working on the WebSpecs of the original one, following the WebTDD cycle and then re-applying the “old” refactorings to the modified specifications when needed.

5 Concluding Remarks and Further Work

In this paper we faced the problem of improving the usability of accessible Web applications. We consider that an application that has been developed to be usable for regular users is generally not usable (even if accessible) for handicapped users, and vice versa. In order to provide a solution for such important problem, we have presented an approach supported by three pillars: a) a catalogue of refactoring specialized in UA problems for blind and visually impaired users; b) a test-driven development process, which uses mockups and Webspecs to simulate the application and to generate the set of tests to assure that all the requirements are satisfied (included the accessibility requirements) and c) a metamodel capable of internally representing the elements of the application and the changes upon these elements (included changes resulting from refactoring) in the same way; this makes easier the evolution of both, the normal application and the UA application. As further work, we are considering how to define catalogues of refactorings for other types of disabilities, for example: hearing impairments, physical disabilities, speech disabilities and cognitive and neurological disabilities. In turn, we are working in order to specialize each catalogue according to the particular type of disability. In addition, the catalogues may be also specialized according to the type of web application: communication applications (facebook, twitter, etc.), electronic commerce (amazon, e-bay, etc.), e-learning, etc. On the other hand, we are considering how to gather and represent usable and accessibility requirements. In this way, the UA refactorings could be applied at any iteration of the development cycle. For this to be feasible (and not too costly), we need to improve the change effect management, to automate the propagation of most changes from the original application to the UA one. Finally, we are improving our tool support to simplify evolution when new requirements affect those pages which were refactored during the usability improvement process. In this sense we need to have a smart composition strategy to be able to compose the “new” change objects with those which appeared in the refactoring stage.

Acknowledgements. This research is supported by the Spanish MCYT R+D project TIN2008-06596-C02-02 and by the Andalusian Government R+D project P08-TIC-03717. It has been also funded by Argentinian Mincyt Project PICT 2187.

References

1. Barnicle, K.: Usability Testing with Screen Reading Technology in a Windows Environment. In: Conf. on Universal Usability, pp. 102--109. ACM Press, New York (2000)
2. W3C.: Web Content Accessibility Guidelines 2.0. December (2008), <http://www.w3.org/TR/WCAG20/>
3. Robles Luna, E., Grigera, J., Rossi, G.: Bridging Test and Model-Driven Approaches in Web Engineering. In: Gaedke, M., Grossniklaus, M. (eds.) Web Engineering. LNCS, vol. 5648, pp. 136--150. Springer Verlag (2009)
4. Zajicek, M., Venetsanopoulos, I., Morrissey, W.: Web Access for Visually Impaired People using Active Accessibility. In: Int. Ergonomics Association 2000/HFES, pp. 445--448. San Diego (2000)
5. Ivory, M., Mankoff, J., Le, A.: Using Automated Tools to Improve Web Site Usage by Users with Diverse Abilities. Journal IT & Society. 1, 195--236 (2003)
6. IBM: Watchfire's Bobby, <http://www.watchfire.com>
7. Harper, S., Goble, C., Steven, R., Yesilada, Y.: Middleware to Expand Context and Preview in Hypertext. In: ASSETS'04, pp. 63--70. ACM Press, New York (2004)
8. Fernandes, A., Carvalho, A., Almeida, J., Simoes, A.: Transcoding for Web Accessibility for the Blind: Semantics from Structure. In: ELPUB2006 Conference on electronic Publishing, pp.123--133. Bansko (2006)
9. Bohman, P. R., Anderson, S.: An Accessible Method of Hiding Html Content. In: the International Cross-Disciplinary Workshop on Web Accessibility (W4A), pp. 39--43. ACM Press, New York (2004)
10. Yesilada, Y., Stevens, R., Harper, S., Goble, C.: Evaluating DANTE: Semantic Transcoding for Visually Disabled Users. ACM Transactions on Computer-Human Interaction (TOCHI), vol. 14, pp.14--es. ACM, New York (2007)
11. Plessers, P., Casteleyn, S., Yesilada, Y., De Troyer, O., Stevens, R., Harper, S., Goble, C.: Accessibility: A Web Engineering Approach. In: 14th International World Wide Web Conference (WWW2005), pp. 353--362. ACM, New York (2005)
12. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
13. Garrido, A., Rossi, G., Distant, D.: Model Refactoring in Web Applications. In: 9th IEEE int. Workshop on Web Site Evolution, pp. 89--96. IEEE CS Press, Washington (2007)
14. Medina-Medina, N., Rossi, G., Garrido, A., Grigera, J.: Refactoring for Accessibility in Web Applications. Proceedings of the XI Congreso Internacional de Interacción Persona-Ordenador (INTERACCIÓN'2010), pp. 427--430. Valencia (2010), Spain.
15. Burella, J., Rossi, G., Robles Luna, E., Grigera, J.: Dealing with Navigation and Interaction Requirement Changes in a TDD-Based Web Engineering Approach. In: Agile Processes in Software Engineering and Extreme Programming. LNCS, vol. 48, pp. 220--225. Springer, Heidelberg (2010)
16. Beck, K.: Test-driven development: by example. Addison-Wesley, Boston (2003)
17. Robles Luna, E., Garrigós, I., Grigera, J., Winckler, M.: Capture and Evolution of Web requirements using WebSpec. In: Web Engineering. LNCS, vol. 6189, pp. 173--188. Springer, Heidelberg (2010)
18. Acerbis, R., Bongio, A., Brambilla, M., Butti, S., Ceri, S., Fraternali, P.: Web Applications Design and Development with WebML and WebRatio 5.0. In: Objects, Components, Models and Patterns. LNCS, vol. 11, pp. 392--411, Springer, Heidelberg (2008)
19. Rivero, J., Rossi, G., Grigera, J., Burella, J., Robles Luna, E., Gordillo, S.: From Mockups to User Interface Models: An extensible Model Driven Approach. To be published in Proceedings of the 6th Workshop on MDWE, Springer LNCS (2010)

Capturing and Validating Personalization Requirements in Web Applications

Esteban Robles Luna
LIFIA, UNLP, Argentina
La Plata, Argentina
erobles@lifia.info.unlp.edu.ar

Irene Garrigós
Lucentia Research Group, DLSI,
University of Alicante, Spain
igarrigos@dlsi.ua.es

Gustavo Rossi
LIFIA, UNLP, Argentina
Also at CONICET
La Plata, Argentina
gustavo@lifia.info.unlp.edu.ar

Abstract — Personalization is a key feature to improve user experience in Web applications and therefore many Web engineering approaches allow the specification of some type of personalization when modelling a website. However, these approaches usually neglect the process of capturing and representing personalization requirements, thus not considering them when the application evolves; maintenance of these requirements is then a very complex task. In this paper, we present WebSpec, a requirement artefact used to capture navigation, interaction and interface aspects of Web applications. Concretely, we focus on how to specify personalization requirements, and on how to automatically generate the personalization model from their specification. Furthermore, from the requirements specification we derive a set of interaction tests to assess the personalization functionality. We illustrate our ideas with an E-commerce application example and describe a prototype tool which implements the described functionality.

Keywords: *Personalization, Web requirements, Requirements Validation*

I. INTRODUCTION

The World Wide Web has changed the way we communicate and exchange information. Web applications have become more complex and the information they provide is continuously growing. Web engineering approaches [2], [4], [7], [9], [12], [20] appeared to provide a systematic way to develop complex Web applications. In this area, personalization [11] has been proposed as a solution to improve the user experience by analyzing his context, characteristics and browsing history and changing different aspects of the application according to his needs.

Due to the different needs and goals of the large and heterogeneous audience that a Web application serves, user expectations need to be considered from the beginning of software projects. However as indicated in [5], most Web engineering approaches do not seriously consider the requirements analysis phase, and as a consequence these requirements are barely taken into account when the application evolves. Therefore, the resulting Web applications usually have outdated requirements which makes impossible to test the actual customer's requirements, and there are difficulties to handle fast evolution, which is usually essential in the Web field.

Personalization is also a missing aspect in the requirement elicitation phase; there are few approaches that

allow modelling personalization requirements (see Sect. VI for details). Moreover, usually (personalization) requirements are described informally, thus becoming a problem when we dive into the implementation and validation phases, particularly to assess if (personalization) requirements have been correctly implemented.

To tackle these problems we use an agile approach called WebTDD [18] which has a TDD (Test Driven Development) style of development; however and differently from “conventional” TDD [1], instead of relying on an extreme coding approach, we use models to generate the application. Using models we raise the level of abstraction as the application is automatically derived from them [18]. Our approach incrementally adds requirements to the existing application, following a short development cycle. WebTDD uses a DSL (Domain Specific Language) called WebSpec [17] to specify these requirements.

In this paper, we focus on how to specify personalization requirements and how to use this specification to improve the development process by automating some time-consuming and error-prone tasks. Summarizing, as the contributions of this paper, we show how to:

- Specify personalization requirements using a model-driven style.
- Automatically generate the conceptual models for the personalization functionality of the Web application, thus avoiding manual errors and the mismatch between the requirements and the implementation.
- Automatically generate tests from the requirements specification to validate the personalization functionality in the WebTDD cycle.

The rest of the paper is structured as follows: in Section II we briefly present the WebTDD approach. In Section III we show how personalization is specified in WebSpec and how we automatically derive interaction tests from the requirements specification. Section IV shows how the personalization model is automatically derived from the personalization requirements. Section V describes the implementation of our ideas. Section VI presents some related work and finally Section VII concludes and presents some further work we are pursuing.

II. WEBTDD IN A NUTSHELL

WebTDD is an agile approach which follows a TDD style of development, using models to generate the Web

application. Like most agile approaches, it is based on short development cycles; in each cycle new requirements are added and the application is upgraded incrementally.

The cycle starts by capturing requirements with mockups (stub HTML pages) to agree on the look and feel of the application, and WebSpec diagrams (Step 1 of Fig. 1) to represent navigation and interaction behaviours. WebSpec is a DSL which allows specifying navigation, interaction and user interface aspects in a more formal way (e.g. in comparison with use cases [10]).

Next we automatically derive (Step 2) a set of meaningful tests that the application must pass to satisfy the captured requirements. As in “conventional” TDD, we run them prior to the implementation (Step 3) in order to check that the application does not satisfy the requirements yet. Afterwards, the modelling activities begin (Step 4): we create or enhance a set of models and derive a running application (Step 5). We check whether each requirement has been successfully implemented by running the previous tests (Step 6). If one test fails, we have to go back, tweak the models and derive the application again until all tests pass. The approach continues with the next requirement until the sprint is over. We must notice that WebTDD is independent of the model driven Web engineering approach used for the modeling activities as the core of the process does not depend on the specific modelling artefacts or mechanics [18].

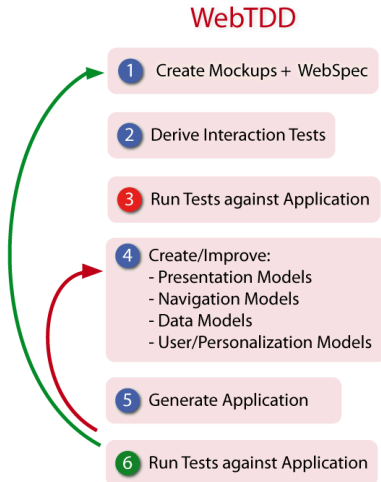


Figure 1. Approach overview

A WebSpec diagram has two key elements: interactions and navigations. An interaction (the counterpart of a Web page in the requirements stage) represents a point where the user can interact with the application by using its interface objects. Interactions may have widgets such as: labels, list boxes, buttons, radio buttons, check boxes and panels. Labels define the content (information) shown by an interaction. A diagram has a starting interaction which is represented with dashed lines. Some actions (clicking a button, adding some text in a text field, etc) might activate a navigation from one interaction to another. These actions are

written in WebSpec’s DSL which conforms to the syntax: `var := expr | actionName(arg1,... argn).`

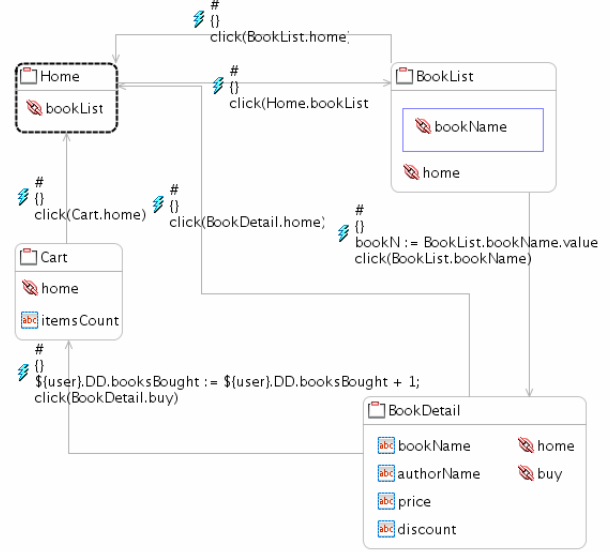


Figure 2. A WebSpec diagram

Fig. 2 shows a WebSpec diagram where navigation in a simplified E-commerce application is specified. The diagram shows how the user can move from one interaction to another thus allowing him to explore books, go back to the home page, buy a book and so on. To express what properties the diagram (and thus the application) must hold, we add invariants to the interactions (invariants are not shown in Fig. 2 for the sake of readability). For instance, the BookDetail interaction must satisfy the invariant `BookDetail.bookName = ${bookN}` which states that the value shown in the bookName label should be equal to the bookN variable (see in Fig. 2 the navigation from BookList to BookDetail where the variable is updated).

After we have specified the scenario in a diagram we can automatically derive a set of tests that the application must satisfy. This is an important feature of WebSpec, because, as in TDD, we use tests as software artefacts that decide whether the requirement is satisfied by the final application. However, instead of the typical unit tests of code-based TDD, we rely on interaction tests which fit better with Web applications. For this scenario, our support tool, the WebSpec Eclipse plug-in (Sect. V), generates a set of tests for the different paths that we can follow from the starting interaction. We next explain how to specify and validate personalization requirements in our approach.

III. SPECIFYING AND VALIDATING PERSONALIZATION REQUIREMENTS

In this section we show how to specify personalization requirements using WebSpec diagrams. Moreover, these requirements can be validated by deriving a set of interaction tests, allowing to check if they are satisfied by the generated application, as explained in Sect. III. B.

A. Specification of Personalization Requirements

A personalization requirement describes some functionality that a Web application has to fulfil to (dynamically) adapt itself, depending on the user or environment profile [11]. In our approach we specify personalization requirements using WebSpec allowing their automatic validation. A WebSpec diagram specifies a personalization scenario that must be satisfied by the final application.

The conditions on which personalization requirements are defined usually refer to user-related information, which is traditionally specified in a so-called user model (UM). This user-related information can be classified in different types:

- User-specific characteristics (independent of the application domain) like age or country.
- Information related to the domain, for instance, from the user browsing behaviour we can derive the preferences or interests on different elements of the domain.
- Information related to the user context (e.g. device, network, actual location, etc).

In WebSpec we use a special variable named $\{user\}$ to denote the different elements associated with the UM. Since, during the requirement elicitation phase the UM does not exist, we assume that the $\{user\}$ variable is a prototype [14] on which we can add properties simply by accessing it and assigning it a value (e.g. $\{user\}.age := 32$). To refer to user-specific characteristics or user-context information, we directly access the property of the user variable, e.g. $\{user\}.age$. In the case of domain dependent information we add the DD prefix, e.g. $\{user\}.DD.booksBought$.

The personalization actions can be specified over the content, the navigation or the presentation of the Web application. Though personalization of the presentation is out of the scope of this paper, we can specify this kind of requirements by associating mockups to interactions (which is usual in WebSpec). Concretely, the personalization actions that can be specified in WebSpec are the following:

- Updating user information: In WebSpec we can specify updates on attributes of the UM by adding actions to the *navigations* of a diagram. The syntax is $\{user\}.attribute := value$ where the value can be a literal or a formula.
- Filtering contents of the site: In WebSpec the labels of the different *interactions* can be filtered according to a condition. This is specified by means of invariants associated to the *interactions* of a diagram. To indicate if a label is shown or not, we use the “visible” property. The syntax is as follows: $label.visible <--> (Boolean\ expression)$. The Boolean expression can also contain a loop, depending on the condition we want to express.
- Filtering the navigation: The links to be shown can also be selected by means of the “visible” property, by specifying invariants over the *interactions* of a diagram. The syntax is as follows: $link.visible <--> (Boolean\ expression)$.

In order to illustrate the described concepts, let’s consider a simple E-commerce application in which our stakeholders want to personalize the discounts offered to customers, depending on how many books they have already bought. In particular, we would like to offer discounts in the book detail page when the user has already bought 2 or more books.

Following the approach (Fig. 1), we start capturing the requirements using WebSpec diagrams. This personalization requirement implies that the application must perform at least two actions. First, it must record how many books the user has already bought, and then it has to show the discount information in the book detail page, depending on how many books he has already bought. The first action is performed when the user navigates from the BookDetail to the Cart interaction (Fig. 2). The navigation has the side effect of adding the book to the shopping cart and thus incrementing the books that the user has already bought. We express it in the action of the navigation as follows:

```
 $\{user\}.DD.booksBought := \{user\}.DD.booksBought + 1$ 
```

This information is domain dependent, so the prefix DD is added to the attribute to update it as explained before. The second action is expressed in the invariant of the BookDetail interaction. The invariant relates the visible attribute of the label and a condition that must hold to let it be visible:

```
BookDetail.discount.visible <-->
( $\{user\}.DD.booksBought \geq 2$ )
```

Concretely, the discount label is visible if the user has already bought 2 or more books.

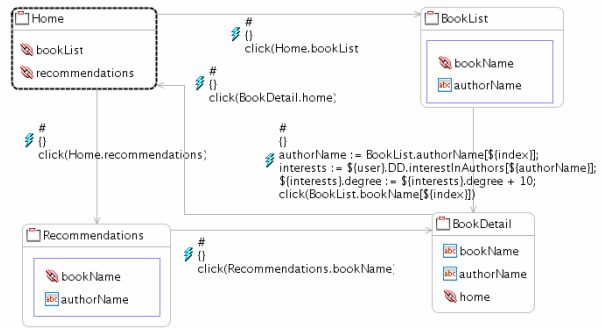


Figure 3. Recommendations personalization scenario

Another example of personalization is a recommendations feature (see Fig. 3); we would like to recommend books of those authors that the user is interested in, using his browsing history. For example, if the user has visited many books of Jose Luis Borges, we could guess that he is one of his favourite authors. This requirement needs first to decide how the users’ interest is captured. We decide to increase the degree of interest when the user navigates to the book details:

```
authorName := BookList.authorName[${index}];
interests :=
{user}.DD.interestInAuthors[${authorName}];
{interests}.degree := {interests}.degree + 10;
```

```
click(BookList.bookName[${index}]))
```

The first action gets the author name. Then we retrieve the information of the interest of the user in the author (interestInAuthors) from the domain dependent information and increase it in 10. Finally, we click on the book's name to move from the BookList to the BookDetail interactions. These 4 actions store the activity of the user that can be later used to show / hide its favourite authors.

Additionally this requirement requires hiding the link that points to the recommendations node when we do not have enough information about the user's interests. So, we specify its visibility in the Home's invariant in this way:

```
Home.recommendations.visible <--> (Exists a in
${user}.DD.interestInAuthors / a.degree >= 100).
```

The above specification states that if there is an author that the user is interested in (degree > 100) then we should show the recommendations link.

B. Derivation of Interaction tests for Requirements Validation

After a requirement has been specified by means of WebSpec diagrams, we are able to automatically derive meaningful interaction tests to assess whether the requirement has been successfully implemented (see Fig 1, step 2). An interaction test opens a Web browser and executes a set of actions in the same way a user would do it. Interaction tests allow making assertions on HTML elements based on XPath expressions so we can check the values of the different widgets.

For each diagram, we create a test suite. Each path depicted in the diagram will be translated into a test case that will be named as the complete path's trail. A test case will follow the actions specified in the path, and assertions will be generated from the invariants of every interaction. The actions specified on navigations will be translated into sentences in the test, for example typing text into a text field or clicking buttons. Reaching an interaction will require that we check its invariant (if any), by generating assertions on the test. As different interactions may alter the variables bound to an invariant, it may be necessary to repeat the updated assertions after navigating to the same interaction more than once.

For example, the discount personalization diagram (see Fig. 2) is derived into the following interaction test (in Selenium [21]). Line 1 opens the application. Lines 2-11 add 2 books to the cart and assert that the discount is not present yet. Lines 12-14 navigate to the book detail page and validate that the discount is present (because the user has already bought 2 books).

```
(01) selenium.open(
      "http://localhost:8080/bookstore");
(02) selenium.click("id=bookList");
(03) selenium.click("id=book1");
(04) assertFalse(selenium.isElementPresent(
      "id=discount"));
(05) selenium.click("id=buy");
(06) selenium.click("id=home");
```

```
(07) selenium.click("id=bookList");
(08) selenium.click("id=book2");
(09) assertFalse(selenium.isElementPresent(
      "id=discount"));
(10) selenium.click("id=buy");
(11) selenium.click("id=home");
(12) selenium.click("id=bookList");
(13) selenium.click("id=book3");
(14) assertTrue(selenium.isElementPresent(
      "id=discount"));
```

After the test derivation process is completed we can run the tests to ensure that the application does not satisfy the requirement yet (Step 3); the same tests will be run again when the requirements have been implemented. The personalization model (Step 4) will be automatically derived from the WebSpec diagrams as shown in the following section.

IV. AUTOMATIC GENERATION OF THE PERSONALIZATION MODEL

Once the personalization requirements have been specified and the tests have been generated, we focus on how to automatically derive concrete software artefacts that implement the personalization functionality from the personalization requirements. In this way, the mismatch between requirements and the developed application is avoided. The generation of such software artefacts leads to an application that satisfies the personalization requirements expressed in the WebSpec diagrams.

In this case, the software artefacts generated from the personalization requirements are personalization rules. We have chosen to specify these rules using the PRML (Personalization Rules Modelling Language) language [7]. PRML is a rule-based high level language devised to specify personalization in an orthogonal way upon Web applications, independently of the underlying methodology. PRML has been successfully used in several Web methodologies and applied to several Web systems and an engine to perform and validate these rules has been implemented [7].

In the following subsections we present how to derive the PRML rules from the WebSpec specifications in a formal way. We also show an intuitive example of PRML rule generation, and finally we explain how to build the UM from the personalization rules.

A. Deriving PRML rules

By automatically generating the personalization model, we provide the designer a first set of personalization rules that he can refine or modify later. This helps avoiding many manual errors and inconsistencies. In order to transform WebSpec diagrams into PRML rules, we use the MOF 2.0 Query/View/Transformation language (QVT) [15] which is a standard transformation language in the context of the MDA (Model Driven Architecture) initiative. QVT is the means for defining formal and automatic transformations between models. Defining transformations by specifying QVT relations has several advantages: (i) transformations are formally established, easy to understand, reuse and maintain, (ii) they do not have to be manually performed by an expert,

which is a tedious and time-consuming task, and (iii) relations can be easily integrated into an MDA approach.

The objective of QVT is to define a formal mapping of the elements of a source metamodel (e.g. WebSpec) into a target metamodel (e.g. PRML). The PRML metamodel can be checked at [7] and the WebSpec metamodel is shown in Fig.4.

The generation of a PRML rule from a WebSpec diagram is defined by a sequence of transformations (QVT relations). A PRML rule is derived from a set of actions specified in WebSpec diagrams. As PRML rules are event-condition-action rules, each of these three parts should be derived from WebSpec specifications:

Depending on the type of WebSpec interaction performed by the user (e.g. navigation, diagram setup actions, etc), we can generate the different PRML events.

- PRML conditions are automatically translated from WebSpec conditions.
- The *actions* of PRML rules are derived by taking into account the different expressions specified in each of the actions of a WebSpec diagram. For instance, we can derive a PRML setContent action (which updates the user information in the UM) from an assignment expression in WebSpec. We can derive actions which filter the attributes to be shown or the links (e.g. selectAttribute and hideLink in PRML) by checking the “visible” attribute of the WebSpec *WidgetReference* element of the metamodel.

Due to space limitations, we cannot show all the QVT rules we have defined. In Fig. 4, the QVT rule for deriving the PRML SetContent action is shown as an illustration. This relation checks that there is a set of elements in the WebSpec action that represents an assignment expression according to the WebSpec metamodel (see Fig. 5). These elements are: an

assignment class together with the corresponding variable to assign the value, and the value (e.g. an expression) to be assigned. The relation enforces that the corresponding PRML expression has the following elements: a setContent class, and an expression that expresses the assignment of a value to a UM variable.

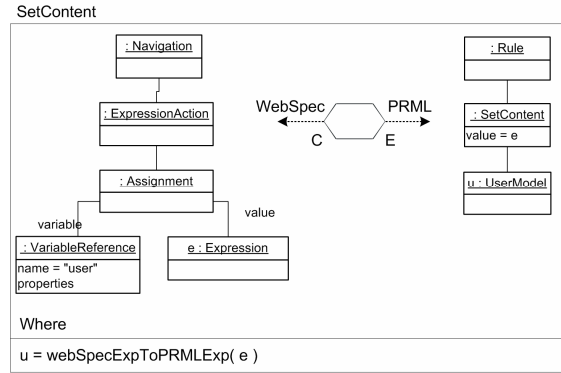


Figure 4. SetContent QVT transformation

To intuitively illustrate the rule generation process, let us consider the discount requirement example explained before (Fig. 2). As aforementioned (Sect. III), this requirement is derived into two PRML rules. The first one (i.e. acquisition rule) acquires/updates the number of books bought by the user in the UM. The second one (i.e. personalization rule) shows/hides the discount attribute to the user based on the previously acquired information (i.e. books bought).

The acquisition rule determines the moment (navigation), condition (always) and the action (increase the value of the variable in the UM). Then, from the navigation in the WebSpec diagram (see Fig. 2) we derive the following PRML rule:

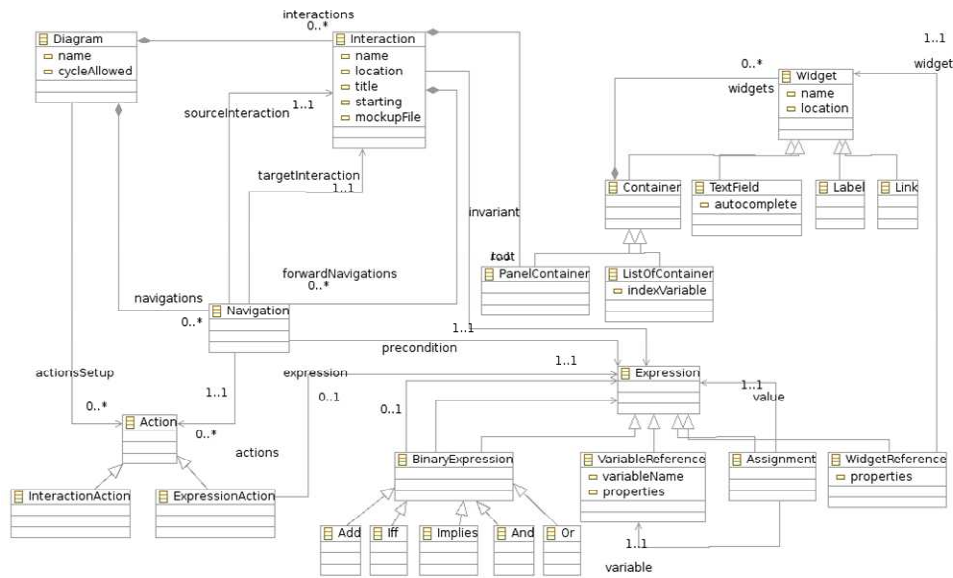


Figure 5. WebSpec's metamodel


```

When Navigation.BookDetailBuy(NM.Book book) do
    setContent(UM.User.booksBought,
        UM.User.booksBought + 1)
endWhen

```

In a similar way, we derive the personalization rule from the BookDetail invariant (see Sect. III). Since personalization takes place every time the node is loaded, the PRML event derived is LoadElement. The condition corresponds with the right part of the WebSpec iff Boolean expression, and the selectAttribute action matches the left part of the iff because it references the visible property of a label. The PRML rule derived is shown next:

```

When LoadElement.BookDetail(NM.Book book) do
    If (UM.User.booksBought >= 2) then
        book.Attributes.selectAttribute(discount)
    endIf
endWhen

```

In the following section we show how we incrementally implement the UM using the derived PRML rules as a starting point.

B. Incremental Implementation of the UM

In the previous section we showed how a set of personalization rules in the PRML language are derived. These rules express the Event-Condition-Actions that have to occur to personalize the application. Since we are deriving these rules from the requirements following a top down process, the UM may not reflect yet the functionality expressed on them. For instance, the first time we derive the rules, the User class may not even exist. Additionally, when the application has been deployed the UM may not reflect a new attribute that has been added by a new requirement. All

these problems are detected by the PRML engine [7] when it validates the generated rules. The validation process will fail showing which parts of the UM do not exist yet.

Using the same philosophy of TDD, we create/enhance the UM in an incremental way by trying to validate the derived rules. The validation process will show which information is not yet present in the UM. For each attribute or class that does not exist in the UM, we create it manually and run the validation process again until the validation succeeds. In this way, we drive the development of the UM using the rules that were automatically generated in the previous step making it a straightforward process.

As an example, let us consider the first rule of the previous subsection. Assuming that the User class already exists, we run the PRML rule validation which fails because the booksBought attribute does not exist in the User class. To make the validation pass, we go to the class and add the instance variable of type number. Then, we run the validation again and finally the validation will succeed.

V. IMPLEMENTATION

WebSpec has been implemented as an Eclipse plugin (Fig. 6) using EMF and GMF technologies. It supports the specification of personalization requirements by means of diagrams that the user can create within the environment and using the palette on the left side of the diagram editor, the user can create concepts like Interactions and Navigations and complete the diagram with the personalization specification.

The automatic derivation of interaction tests is performed using a JUnit class writer that satisfies the syntax needed by the Selenium framework. Also, during test derivation, expressions are optimized for better readability. For

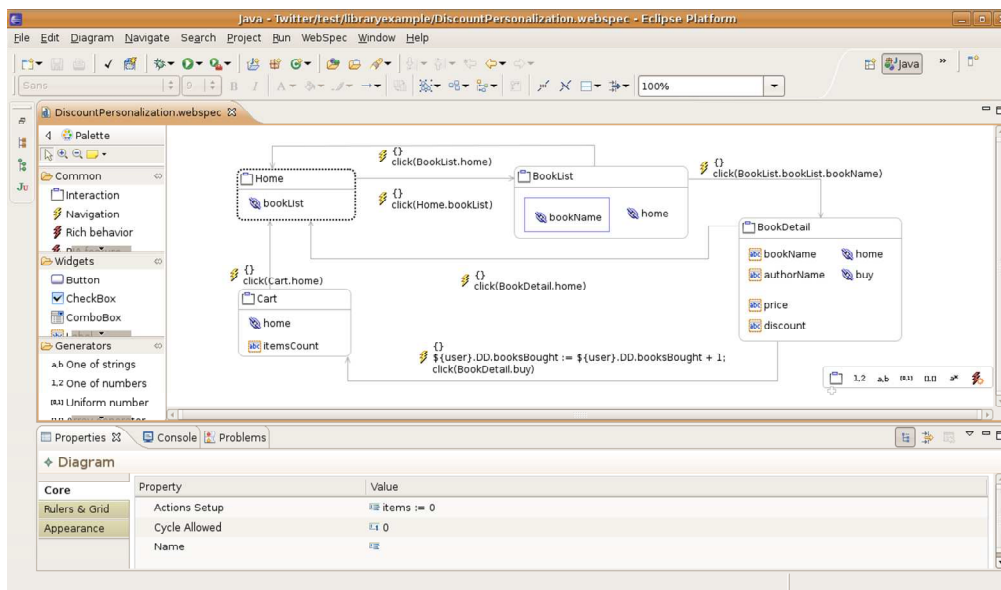


Figure 6. WebSpec's Eclipse plugin

example, an expression like: `{long} -> Home.username = "John"` where the long variable has the value false (`->` means the implies relationship) is automatically optimized to true using Boolean equivalencies. We have chosen JUnit and Selenium because they are easily integrated in Eclipse though other web testing framework such as Watir [22] can also be used.

The automatic derivation of PRML rules is easily performed as PRML is also implemented as an Eclipse plugin thus allowing to seamlessly integrate both approaches. The WebSpec menu has options to allow the derivation of PRML rules that are automatically imported in the PRML prototype tool. PRML rules are plain text files thus the generation of such rules is easily performed by a model to model transformation from the WebSpec's metamodel to the PRML metamodel. Then we reuse the transformation process of the PRML tool to use the model to text transformation.

We have used the WebSpec plugin with the PRML tool to implement a personalization version of the E-commerce application. Several personalization requirements have been specified and validated using the derived tests in the context of the WebTDD approach. We have used interaction tests to drive the development of the personalization and functional requirements. Tests were used to check that the new requirements have been correctly implemented and that we have not been unintentionally broken existing functionality. The personalization model was derived from the specification thus avoiding the mismatch between requirements and the implementation. However, as previously mentioned, we have to follow a short TDD cycle to complete the derivation as it only covers some structural aspects of classes in PRML. We expect to improve the derivation process in future work.

VI. RELATED WORK

In the context of Web engineering, few approaches have focused on defining an explicit requirement analysis stage to model the user expectations. Some approaches consider the modelling of personalization to some extent [3], [8], [9], [12], [20]. In general those approaches ignore how personalization requirements are captured.

A-OOH [8] is a model-driven approach which allows the specification of personalization requirements. It uses the *i** framework in order to specify a goal-oriented requirements model. From this specification, the conceptual models (e.g. domain and navigation models) are generated by means of QVT transformations. However, A-OOH does not allow the derivation of the personalization model as done in WebSpec.

In [13], in the context of OOHDM [20], personalized UIDs are used to capture a personalized version of the interactions that users have with the application. The difference with traditional UIDs is that they may have many initial interactions one for each different type of user. WebSpec and personalized UIDs share the same terminology as WebSpec is based on UIDs; however, personalized UIDs do not provide automatic transformations to software artefacts, so there may be a mismatch between requirements and the final application.

Adaptive OOWS [19] extends OOWS [16] to support adaptation. It proposes two artifacts to specify adaptive requirements: an enhanced of Activity Diagrams called User Stereotype Diagrams and their corresponding User and Data Specifications descriptions which capture the adaptive part of the requirements by means of intuitive and easy-to-understand schemas. Afterwards, the requirements models serve as a basis to derive the conceptual specifications of users and adaptive features in the OOWS Conceptual Modeling phase. This approach shares some common features with the work presented in this paper such as: specification of requirements and derivation of the User model. However, the approach does not provide automatic ways to validate that the adaptive requirements are correctly implemented in the application. Requirements validation is extremely important to ensure that the behavior of the application is preserved, e.g. when maintainability needs to be improved by means of model refactorings.

In [6], Escalona and Koch have proposed a metamodel based on WebRE profiles to specify web requirements. Its main advantage is the automatic generation of conceptual models (content and navigation models) which automatically satisfy the requirements. Also, some tests are derived from the profiles to validate that the functionality has been correctly implemented. However, some requirements such as detailed composition of the user interface and specifically personalization requirements can not be specified thus requirements cannot be validated and the personalization models can not be derived using this notation.

In summary, the described approaches are, as far as the authors are concerned, the only that allow specifying personalization requirements, however they have the following drawbacks:

- They do not allow the automatic derivation of the personalization artefacts (personalization and UM). Doing so we avoid many manual errors and we assure that the defined model is aligned with the previously specified requirements.
- They do not provide a way to validate personalization requirements. Automatic validation using tests helps not only to validate the correct implementation of the personalization requirements, but it also helps to detect unintended errors when the application grows.

WebSpec supports the specification of Personalization requirements and can be used in different development processes to implement the personalization functionality. To the authors' knowledge, the work presented in this paper is the first to provide test derivation and partial UM derivation from a requirement artefact specifically for Personalization. In addition to the advantages shown in this work, we can use WebSpec in conjunction with mockups to improve the communication between stakeholders while capturing the personalization requirements as shown in [17].

VII. CONCLUSIONS AND FURTHER WORK

In this paper we have presented an approach for dealing with personalization requirements in Web applications. Requirements are captured in WebSpec diagrams which

allow us to derive a set of tests to validate requirements, and to automatically derive the personalization rules in the PRML language. In addition, we have shown how the UM can be incrementally implemented by validating the generated rules in the PRML engine. The idea has been presented in the context of WebTDD, an agile approach for developing Web applications, but it can be applied to any other Web methodology.

We are currently working on the automatic derivation of the UM which is, until now, done manually (as shown in Sect. IV B). Furthermore, we are working on some field experiences with the usage of mockups to help on developing the look and feel of the personalization functionality. Finally, we are analyzing how personalization requirements evolve and how we handle this evolution along the development cycle.

ACKNOWLEDGEMENTS

This work has been partially supported by the MANTRA project (GRE09-17) from the University of Alicante, and by the MESOLAP (TIN2010-14860) from the Spanish Ministry of Education and Science.

REFERENCES

- [1] Beck, K.: Test Driven Development: by Example, Addison-Wesley, 2003.
- [2] Casteleyn, S., Garrigós, I., Troyer, O.D.: Automatic runtime validation and correction of the navigational design of web sites. In: APWeb. (2005) 453–463.
- [3] Ceri, S., Daniel, F., Matera, M., and Facca, F. M. 2007. Model-driven development of context-aware Web applications. *ACM Trans. Internet Technol.* 7, 1 (Feb. 2007), 2.
- [4] Ceri, S., Manolescu, I.: Constructing and integrating data-centric web applications: Methods, tools, and techniques. In: VLDB. (2003) 1151.
- [5] Escalona, M.J., Koch, N.: Requirements engineering for web applications – a comparative study. *J. Web Eng.* 2(3) (2004) 193–212.
- [6] Escalona, M.J., Koch, N. Metamodeling Requirements of Web Systems. In *Proc. International Conference on Web Information System and Technologies (WEBIST 2006)*, INSTICC, 310–317, Setúbal, Portugal. 2006.
- [7] Garrigós, I.: A-OOH: Extending Web Application Design with Dynamic Personalization. PhD thesis, University of Alicante, Spain (2008)
- [8] Garrigós, I., Mazón, J.N., Trujillo, J.: A Requirement Analysis Approach for Using i* in Web Engineering. In: ICWE. (2009), LNCS, 5648, 151–165.
- [9] Houben, G.-J., Frasincar, F., Barna, P. and Vdovjak, R. (2004). Engineering the presentation layer of adaptable web information systems. In *Web Engineering 4th International Conference, ICWE 2004*, volume 3140 of *Lecture Notes in Computer Science*, pages 60–73, Springer, ISBN 3-540-22511-0.
- [10] Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press/Addison-Wesley, 1992.
- [11] Kim, K: Personalization: Definition, Status, and Challenges Ahead. *Journal of Object Technology*, 1, (2002) 29–40.
- [12] Koch, N.: Reference model, modeling techniques and development process software engineering for adaptive hypermedia systems. *KI* 16(3) (2002) 40–41.
- [13] Martin, A. Cechich, A: A Model-Driven Reengineering Approach to Web Site Personalization. In *Proceedings of the Third Latin American Web Congress (October 31 - November 02, 2005)*. LA-WEB. IEEE Computer Society, Washington, DC, 14.
- [14] Noble, J., Taivalsaari, A., Moore, I. (eds.): *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag. ISBN 981-4021-25-3. 1999.
- [15] QVT Language: <http://www.omg.org/cgi-bin/doc?ptc/2005-11-01>.
- [16] Pastor, O., Abrahão, S., Fons, J.: An Object-Oriented Approach to Automate Web Applications Development. In: Bauknecht, K., Madria, S.K., Pernul, G. (eds.) *ECWeb 2001*. LNCS, vol. 2115, pp. 16–28. Springer, Heidelberg (2001).
- [17] Robles Luna E., Garrigós I., Grigera J., Winckler M. Capture and Evolution of Web requirements using WebSpec. To be published in the *Proceedings of 10th International Conference on Web Engineering (ICWE 2010)*.
- [18] Robles Luna, E., Grigera, J., Rossi, G.: Bridging Test and Model-Driven Approaches in Web Engineering, *Web Engineering, Lecture Notes in Computer Science*, pp. 136–150, Springer, Heidelberg, June 2009.
- [19] Rojas, G., Valderas, P., and Pelechano, V. 2006. Describing Adaptive Navigation Requirements of Web Applications. In *Proc. of the 4th International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems (AH 2006)*, Dublin, Ireland. LNCS 4018. 318–322.
- [20] Schwabe, D., Rossi, G.: An object oriented approach to web-based applications design. *TAPOS* 4(4) (1998) 207–225
- [21] Selenium, a Web application testing system, <http://seleniumhq.org/>
- [22] Watir, <http://watir.com/>

Change management and tool support for WebSpec

The content of this chapter corresponds with the following papers:

*Burella J., Rossi G., **Robles Luna E.**, Grigera J. Dealing with Navigation and Interaction Requirement Changes in a TDD-Based Web Engineering Approach. Proceedings of the 11th International Conference on Agile Software Development (**XP 2010**), Springer Verlag, LNCS, 2010. Trondheim, Norway. Core B.*

***Robles Luna E.**, Burella J., Grigera J, Rossi G. A Flexible Tool Suite for Change-Aware Test-Driven Development of Web Applications. Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (**ICSE 2010**). 2010. Cape Town, South Africa. Core A.*

In the previous chapters we have shown how to use WebSpec with WebTDD and in the context of a methodology that uses i*. Also, we showed its use not only for specifying functional requirements but also for personalization and accessibility ones.

Though we have briefly presented in the previous chapters some details about WebSpec tool, in this chapter we concentrate in WebSpec's change management support and its Eclipse tool.

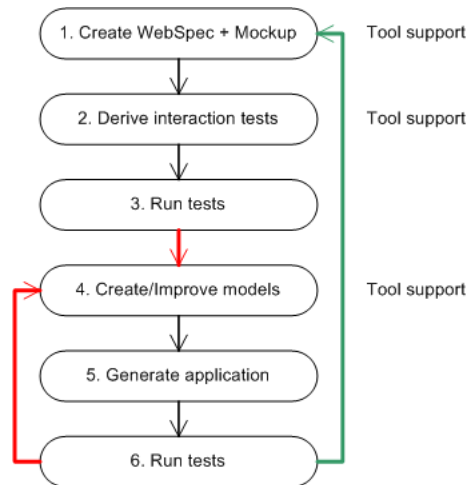


Fig. 7.1. Tool support of WebSpec in WebTDD

The content of this chapter corresponds with a paper published in the *extreme programming conference (XP)* and with a research demo presented in the *International Conference on Software Engineering (ICSE)*. XP is a leading international conference on agile methods in software and information systems development with the aim to bring together software and information systems professionals, both researchers and practitioners, to discuss the latest trends, applications, and theory, share experiences, and reveal new research results in agile software development. On the other hand, ICSE is the premier international event for software

engineering. It provides a world class forum for software engineering professionals from industry, government and academia to hear about and discuss the latest developments, trends and innovations in software engineering.

Dealing with Navigation and Interaction Requirements Changes in a TDD-Based Web Engineering Approach

Juan Burella^{1,3}, Gustavo Rossi^{2,3}, Esteban Robles Luna^{2,3}, Julián Grigera²

¹Departamento de Computación, Universidad de Buenos Aires
jburella@dc.uba.ar

²LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
[gustavo, esteban.robles, julian.grigera]@lifia.info.unlp.edu.ar

³Also at CONICET

Abstract. Web applications are well suited to be developed with agile methods. However, as they tend to change too fast, special care must be put in change management, both to satisfy customers and reduce developers load. In this paper we discuss how we deal with navigation and interaction requirements changes in a novel test-driven development approach for Web applications. We present it by indicating how it resembles and differs from “conventional” TDD, and showing how changes can be treated as “first class” objects, allowing us to automate the application changes and also to adaptively prune the test suite.

1 Introduction

TDD and its variants like STDD [4] mostly focus on behavioural aspects of domain classes, and since TDD is generally applied in a bottom up way, it tends to disregard important features of Web applications such as navigation, interface or interaction. As a consequence, usability, look and feel, and also navigation features may be checked too late, once the application has been already presented to the customers, thus delaying the correction process.

As a way to overcome the mismatch between “conventional” TDD and Web applications development, we present an approach for improving change management in the context of our TDD-like methodology by focusing on changes that affect navigation and interaction aspects. Requirements are represented using WebSpec diagrams, which capture navigation, user interface (UI) and interaction application aspects. WebSpec diagrams are then automatically translated into sets of meaningful interaction tests the application must pass. While the developers work coding the solution, the support environment captures the changes in objects and associates them to the corresponding tests. Change objects can also help to semi automatically change structural parts of the Web application when a requirement is added or changed. In the same way, we can reduce the number of tests that must be run to those that exercise changed objects only, improving the overall development time. We illustrate the approach with a simple Twitter-like application and show an integrated environment built on top of Seaside (www.seaside.st) that supports this functionality.

2 Background: A Test-Driven Approach for Web Applications

The key aspects of our requirements modelling stage are fast interface and interaction prototyping on one hand, and navigation modelling on the other. Prototyping is carried out with interaction mockups: simple HTML stub pages that significantly help to agree on the application's look and feel, and the way interaction must be performed.

We use WebSpec diagrams to specify navigation and interaction requirements more formally than with User Stories (US) [3]. These artefacts (based on UIDs [5] and Quickcheck [1]) capture navigation and interaction aspects in a similar way UIDs do, but adding the formal power of preconditions and invariants to assert properties in the interactions. A WebSpec diagram contains *interactions* and *navigations*. An *interaction* represents a point where the user consumes information (expressed as a set of interface widgets) and interacts with the application by using some of its widgets. Some actions (clicking a button, adding some text in a text field, etc) might produce *navigation* from one *interaction* to another, and as a consequence, the user moves through the application's navigation space. These actions are written in an intuitive domain specific language. Fig. 1 shows a WebSpec that will let the user tweet, see how many tweets he has, and allow him to logout from the application. From the Login *interaction*, the user can authenticate by typing its username and password and then clicking on the login button (navigation from Login to Home interaction). Then, the user can add messages by typing in the messageTF and clicking on the post button (navigation from Home to Home interaction).

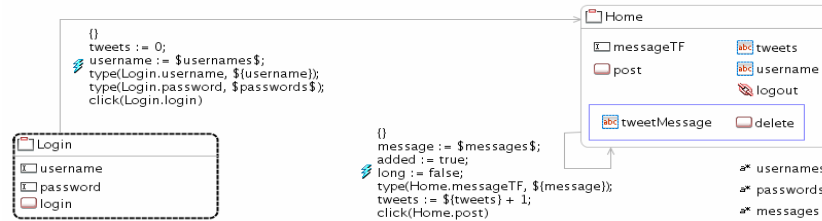


Figure 1. WebSpec of Tweet's interaction

From a WebSpec diagram we automatically generate a set of interaction tests that cover all the interaction paths specified in it, thus avoiding the translation problem of TDD between tests and requirements. Unlike unit tests, interaction tests simulate user input into HTML pages, and allow asserting conditions on the results of such interactions. Since each WebSpec *interaction* is related to a mockup, each test runs against it and the predicates are transformed into tests assertions. These (failing) series of tests set a good starting point for our TDD-like approach.

Once we have a set of tests for a specific requirement, it is time to develop the functionality to make them pass. Since interaction tests define the functionalities at user level, and we will drive the development from such tests, our approach will naturally follow a top-down style, rather than the usual bottom-up way of regular TDD. Nevertheless, we will use regular TDD as we dive into the application's underlying domain model.

Basing ourselves in the mockups, we recreate the same UI using widgets, but this time adding stubs for the dynamic behaviour in the places where the application will

interact with the domain objects. Next, for every stub message left in the presentation and navigation classes, we code the model classes to fill the gaps.

At the time we engage in the development of the domain model classes, we follow a traditional TDD cycle by creating the unit tests first to establish the purpose of the new objects, which is in turn facilitated by the UI/navigation models which have already set specific requests for them. Once unit tests pass, we can run the interaction tests to check whether we have completed the necessary functionality for the current request. As usual, when tests do not pass, we keep working on the code until they do, and once this happens we can go on with the next requirement.

At times, some domain behaviour is needed, and it is not possible to state it as a UI requirement triggered by the user, or cannot be validated at the interaction level. In such cases, we capture the functionality using US and then create unit tests.

For the sake of conciseness, we will focus our explanation on navigation and presentation requirements, and therefore we will not talk about the effect of changes in “conventional” unit tests.

3 Our Approach to Change Management in a Nutshell

We borrow ideas from changeboxes [2] to make software changes explicit and manageable. We specifically focus on navigation and interaction changes in Web applications requirements to minimize the effort for satisfying their impact on the implementation. These changes are explicitly represented as first-class objects, and related to the artefacts in which they produce modifications, and as a consequence, we not only obtain better traceability features, but we are also able to automate some of these changes in the final application. Since navigation tests are also represented with objects, and we can determine the elements they access, and we can know exactly which tests are affected by a change. As a result, we can set apart the tests that will not check the new functionality at all, leaving only those that are really needed to check the new change and its consequences.

We have built a support tool that manages change objects and their relationships with the approach’s artefacts. Being developed on top of the Seaside Squeak’s environment (www.squeak.org), it allows to maintain these relationships during the whole life-cycle, helping to dynamically manipulate even application objects.

3.1. Representing Changes as first-class Objects

As we show in Fig. 2, an application is developed by incrementally applying sets of changes (Step 4). Starting from the initial status, a first set of changes is applied in order to get an initial prototype. In further iterations, the application is extended with new sets of changes, fulfilling the requirements one by one. We stress that the main difference with “pure” TDD is that we automatically derive navigation and interaction tests from WebSpec diagrams, we actively use Mockups to derive the final application’s interface, and we use interaction tests to guide the development of the application’s behavior.

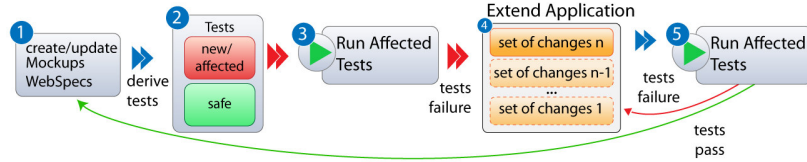


Figure 2. A TDD-based Web Development Process

Fig. 3 illustrates how the approach is improved using change management features. In the first stage, changes made in WebSpecs are captured into change objects. Then, changes made in the application's model are also captured into objects and associated with the corresponding test. We use these objects to reduce the set of interaction tests that drive the upcoming development steps to those affected by them.

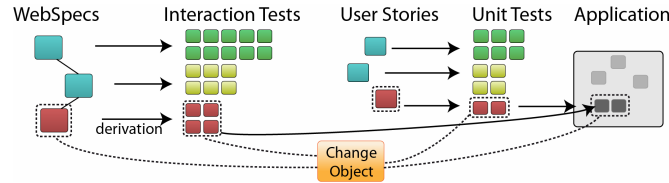


Figure 3. Change Object's relationship with model and tests artefacts

WebSpecs can suffer different coarse grained changes, such as the addition or deletion of an *interaction* or *navigation*. These entities can be modified too, by the addition or deletion of widgets to an *interaction*, changes in invariants, etc. Regarding *navigations*, we can add or delete preconditions, change their source, target, or the action that triggers them. All these types of changes have been represented as classes.

3.2 Mapping requirement changes onto the implementation

Some changes have direct effects on concrete application's artefacts; an important aspect of the corresponding change objects is that they can help to reduce the impact of these changes on the implementation. A *WebSpec* change object is associated to an effect on a Web artefact; this effect is also represented as a change object. These objects are able to produce the real modifications on their targets with the help of an *Effect Handler*. The *Effect Handler* is a component that knows how to perform changes on a concrete platform such as Seaside or GWT. For example, when a change modifies an interaction structurally, the page that represents this interaction must be modified: e.g. when a label is added to an interaction, it adds an equivalent label on the page represented by the modified interaction (Fig. 4).

Regarding *navigation* changes, we can change preconditions, sources, targets, or the actions that triggers them. The first type of change does not generate effects on the final application look and feel; in turn, if the *navigation* target changes, we can automate the effect of this change, for example linking the page associated to the new target *interaction*. Something similar happens when a *navigation* action is changed.



Figure 4. A label addition change in action

4 A Proof of Concept

To illustrate our approach we describe how we put it into work in the Seaside framework, by implementing a specific *Effect Handler* for this platform. In the simplified Twitter-like application presented in Sect. 2, we started with a short sprint to capture the basic user stories: login and tweet. We will only discuss changes related with the tweet use story, and assume that we have finished the first iteration and the application satisfies the requirements captured by the WebSpec presented in Fig. 1.

Let us suppose that our customer wants to add the possibility to navigate from the home to a ‘Terms of Service’ page. In order to satisfy the new requirement, the development iteration starts with the requirements change (Step 1 in Fig. 2). We specify the *interaction* and *navigation* paths that we expect for the ‘Term of service’ requirement, which produces a set of change objects derived from the changes in the WebSpec diagram that express the link creation, the creation of the “terms of service” *interaction*, and the *navigation* between both *interactions* (Fig. 5).

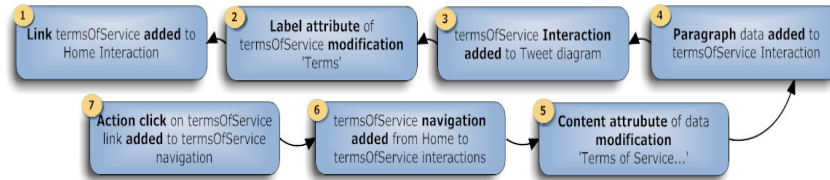


Figure 5. Change objects associated with the “Terms of Service” requirement

We then derive a set of tests from the WebSpec diagram and find that it generates a new test that checks the terms of service navigation, while in the previous iteration we had two tests that checked the addition of valid and invalid messages. To avoid running all three tests, we ask the Change Manager to determine which are affected by this change. As the only affected is the new one (Step 2 in Fig. 2) we run it, and it fails (Step 3 in Fig. 2), thus we must implement the new changes to satisfy this test.

The process continues with the change effect management (Step 4 in Fig. 2) iterating over each change to see how it impacts on the implementation. The first change generates a creation method for the link widget in the *WAHome* class; it represents the page for Home interaction, so it will be drawn each time a *WAHome* instance shows. The next one modifies the widget label attribute to display the correct link name ‘Terms’. Fig. 6 shows these effects. Change number 3 creates the Seaside component *WATermsOfService* that represents the page for this interaction. The next change generates a creation method for the paragraph widget in the *WATermsOfServices* class, and the next one modifies its content attribute, in the creation method. The *navigation* addition change does not produce modifications, but the last change generates the necessary code for associating the *interaction* pages through the “terms” link. Finally,

we run the affected test realizing that it passes because of the semi automatic changes we applied on the application, thus completing the iteration (Step 5 in Fig.2).

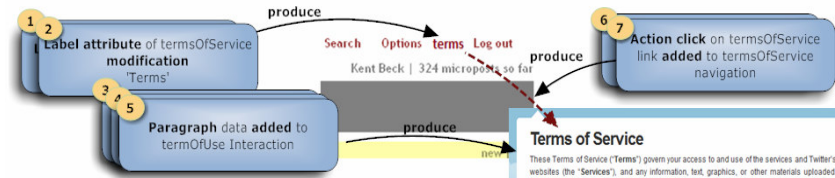


Figure 6. Managing the effects of the “Term of Use” requirement

5 Concluding Remarks and Further Work

In this paper we have presented an approach to deal with navigation and presentation requirement changes in the context of a TDD process for Web applications. Our main strategy has been to reify these changes into “first class” objects, so they can not only capture the history of changes, but also trace the effects of changes in different development artefacts, such as tests and application components.

An integrated tool built on top of the Squeak environment allows us to manipulate these change objects, making them extremely useful in the development process. In particular, we have shown how to help the developer by automating some modifications at the presentation level, or advising him about the necessary changes. At the same time, change objects allow reducing the number of tests that must be run, as they maintain a trace with their corresponding tests. Notice that this kind of change-aware development environment is easier to implement in a reflective system like Squeak, though much harder in Java-based environments such as Eclipse. In this sense, we are working on a light version of our environment for the Eclipse platform.

References

1. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: 5th ACM SIGPLAN international conference on Functional programming, pp. 268-279. ACM, New York (2000).
2. Denker M., Gırba T., Lienhard A., Nierstrasz O., Renggli L., Zumkehr P.: Encapsulating and exploiting change with changeboxes. In: 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007, vol 286, pp. 25-49. ACM, New York (2007).
3. Jeffries, R.: Extreme programming installed. Addison-Wesley, Boston (2001)
4. Mugridge, R.: Managing Agile Project Requirements with Storytest-Driven Development. IEEE software, 25, 68-75 (2008).
5. Rossi, G., Schwabe, D.: Modeling and Implementing Web Applications using OOHDm. In: Web Engineering, Modelling and Implementing Web Applications, pp. 109-155. Springer, Heidelberg (2008).

A Flexible Tool Suite for Change-Aware Test-Driven Development of Web Applications

Esteban Robles Luna
LIFIA. F. Informática, UNLP
La Plata, Argentina
Also at CONICET
erobles@lifa.info.unlp.edu.ar

Juan Burella
DC. F. Cs Exactas, UBA
Buenos Aires, Argentina
Also at CONICET
jburella@dc.uba.ar

Julián Grigera
LIFIA. F. Informática, UNLP
La Plata, Argentina
juliang@lifa.info.unlp.edu.ar

Gustavo Rossi
LIFIA. F. Informática, UNLP
La Plata, Argentina
Also at CONICET
gustavo@lifa.info.unlp.edu.ar

ABSTRACT

Though Web Applications development fits well with Test-Driven Development, there are some problems that hinder its success. In this demo we present a tool suite to improve TDD; the suite supports the representation of web requirements using a domain-specific language and the automatic generation of interaction tests among others.

Keywords

Web engineering, TDD, Web requirements, Change management.

1. INTRODUCTION

Test-Driven Development (TDD) [2] is well suited for Web applications because of its features: it is agile, it uses tests as requirements artifacts, and also uses them to determine what requirements have been fulfilled. However, traditional unit testing fails to provide quick feedback to stakeholders about interaction and navigational requirements (i.e. those that affect look and feel and represent the very nature of most Web applications). Additionally, as navigation and interaction requirements change rapidly and often, there is a need to improve change management to automatically update the test suite and simplify application evolution. By capturing requirement changes and deriving traceability links between requirements and the software components that fulfill those requirements, we can use change objects to upgrade the application under development.

In [4] we presented WebTDD, an improvement of TDD aimed at Web software. Our approach follows the basic TDD principles, but instead of driving the development from handcrafted unit tests, we start the process from automatically generated interaction tests, which capture the way users interact with the application and also help to outline the navigation and business models.

Due to the gap between requirements (e.g. expressed in use stories [3]) and tests, some customer requirements might remain unchecked. To bridge this gap, and considering the nature of Web applications, we have devised a domain-specific language (DSL) called WebSpec. WebSpec is used to capture interaction and navigation requirements. Its diagrams have a two-fold objective: they formalize navigation and interaction requirements, and they

serve to automatically generate a suite of interaction tests that the final application must pass. We complement these diagrams with Mockups (stub HTML pages).

As shown in Fig. 1, WebTDD follows a sprint based process; in each sprint a set of requirements is implemented. We first capture requirements (Sect. 2) and use them to simulate the application. Then, we automatically generate a set of interaction tests (Sect. 3) that the application must pass. When we capture requirements we record the changes (Sect. 4) as first class objects and use them to improve the implementation phase. In this paper we present our tool suite to support WebTDD. Specifically we show:

- How to express navigation and interaction requirements, simplifying the discussion with stakeholders.
- How tests are derived automatically from requirements.
- How changes in requirements are captured and then used to improve the development cycle.

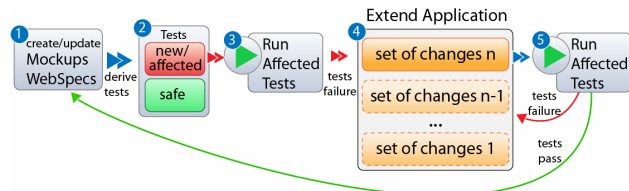


Figure 1. WebTDD approach

2. CAPTURING REQUIREMENTS

The development cycle starts by capturing requirements with Mockups and WebSpec diagrams (Step 1 of Fig. 1). Mockups help to agree on the application look and feel and WebSpec allows to specify navigation, interaction and user interface aspects in a more formal and comprehensive way (than, for example, user stories). A WebSpec diagram can be derived either from use cases or usage scenarios or stories. Similarly, mockups can be created using modern tools like Balsamiq [1]. WebSpec is independent of these technologies as long as the user interface elements can be referenced using an ID based location (e.g. button with id = "search"). WebSpec has two key elements: *interactions* and *navigations*. An *interaction* represents a point where the user can interact with the application by using the *interaction's* widgets. A diagram has a starting interaction represented with dashed lines. Some actions (like clicking a button) might produce *navigation* from one *interaction* to another. These actions are written in an intuitive DSL with the syntax: `var := expr | actionName(arg1, ..., argn)`. We associate a mockup to each *interaction* to allow switching between the formal description and the proposed user interface while discussing with the stakeholders.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

An example of a WebSpec diagram as produced by the tool suite is shown in Fig. 2. To express the properties the application must hold, we add invariants (Boolean predicates) to the *interactions*. For instance, the Home *interaction* (not shown for clarity) must satisfy: *Home.tweets = \${tweets}* which states that the value shown in the tweets label should be equal to the number of tweets variable (see the navigations where the variable is updated).

Using the mockups together with the actions and properties the application must hold, we derive a set of simulations that begin from the starting interaction. Each simulation opens a browser and reproduces a specific path executing actions and showing labels of the expected behavior of the application. Stakeholders can use them to propose changes before the implementation stage.

The WebSpec Eclipse plugin provides an environment to create WebSpec diagrams and to simulate them in a real browser.



Figure 2. Tweet WebSpec diagram

3. REQUIREMENTS VALIDATION

In the 2nd step of Fig. 1 we automatically generate a set of interaction tests from the WebSpec diagram. An interaction test is a test that pops a Browser and executes a set of actions on it, in the same way a user would do. This kind of tests allows making assertions on UI elements based on their location, so we can check the values of the different widgets. We can also automatically verify whether a requirement has been successfully implemented by validating that the application passes all tests.

For each WebSpec diagram, we derive a test suite. Each path depicted in the diagram will be translated into a test case that will be named as the complete path's trail. If the diagram is cycled, we obtain finite paths by pruning to a specific length. A test case will follow the actions specified in the path, and assertions will be generated from the invariants of every *interaction*. The sentences (assignments or actions) on *navigations* will be translated to sentences in the test, such as typing text into a text field or clicking buttons. Reaching an *interaction* will require that we check its invariant (if any), by generating assertions on the test. As different *interactions* may alter the variables bound to an invariant, it is necessary to repeat the updated assertions after navigating to the same *interaction* more than once.

The WebSpec Eclipse plugin supports tests generation to Selenium [5] but other testing frameworks could be easily added by extending the generation algorithm.

4. EVOLUTION

Evolution of applications starts with changes in the requirements, and navigation/interaction requirements changes are specially frequent during the development process. WebSpec can suffer different changes, such as the addition or deletion of an *interaction* or *navigation*. An *interaction* can be modified too by the addition or deletion of widgets, changes in invariants, etc. Regarding *navigations*, we can change its preconditions or the actions

that triggers them. All types of changes have been reified as first-class change objects that could be used to improve the tool's traceability features and automate some of these changes in the implementation. The WebSpec editor captures the changes made to the diagrams and stores them in files to be latter use.

To improve the development cycle (Step 4 of Fig. 1) the suite includes a change management tool that allows the manipulation of these change objects to automate the effects of changes on concrete application's artifacts. The mechanics of these effects depend on the underlying implementation setting (GWT, WebRatio [6], etc) thus we have handlers for each particular case.

As an example, let us suppose that an interaction has been added to a WebSpec, so we create the corresponding Web page for this *interaction*. When a change modifies an interaction structurally, the page that represents this interaction must be modified accordingly; Figure 3 shows how a text field element is added as the effect of the addition of a text field in the interaction. If any widget attribute is changed, the effect on the page can be automatically updated too. The tool suite has effect handlers for GWT, Seaside and WebRatio, but new ones can be easily implemented.

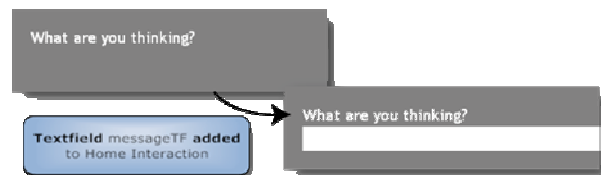


Figure 3. Text field added to the Home interaction

5. CONCLUSIONS

We have shown an agile approach for Web applications development and briefly described its supporting tool suite. Our WebTDD tool suite allows us to visually specify navigation and interaction requirements, automatically simulating the application according to requirements and generating navigation tests to validate these requirements. Changes are reified using "first class" objects, and using a change management tool we can manipulate these change objects, making them very useful in the development process. We have shown how to improve the development process by automating the effect of presentation and navigation changes.

6. REFERENCES

- [1] Balsamiq. <http://www.balsamiq.com/products/mockups>
- [2] Beck, K. 2002 Test Driven Development: by Example. Addison-Wesley Longman Publishing Co., Inc.
- [3] Jeffries, R. E., Anderson, A., and Hendrickson, C. 2000 Extreme Programming Installed. Addison-Wesley Longman Publishing Co., Inc.
- [4] Robles Luna, E., Grigera, J., and Rossi, G. 2009. Bridging Test and Model-Driven Approaches in Web Engineering. In Proceedings of the 9th international Conference on Web Engineering. Lecture Notes In Computer Science, vol. 5648. Springer-Verlag, Berlin, Heidelberg, 136-150.
- [5] Selenium web testing system. <http://seleniumhq.org/>
- [6] The WebRatio Tool Suite. <http://www.Webratio.com>.

Appendix: Papers Already Submitted

A

WebSpec: a Visual Language for Specifying Interaction and Navigation Requirements in Web Applications

*The content of this chapter corresponds with the following paper:
Robles Luna E., Rossi G., Garrigos I. *WebSpec: a Visual Language for Specifying Interaction and Navigation Requirements in Web Applications*. *Requirements Engineering Journal*. In press.
Impact factor: 0.931. JCR.*

In this chapter we present an evolution of the core language presented in chapter 4 in which we detail the specification of requirements for rich internet applications. We show how to specify some patterns that appear often in Web applications easily and provide the language's grammar. Also, we provide extensions to the case of study presented in chapter 4.

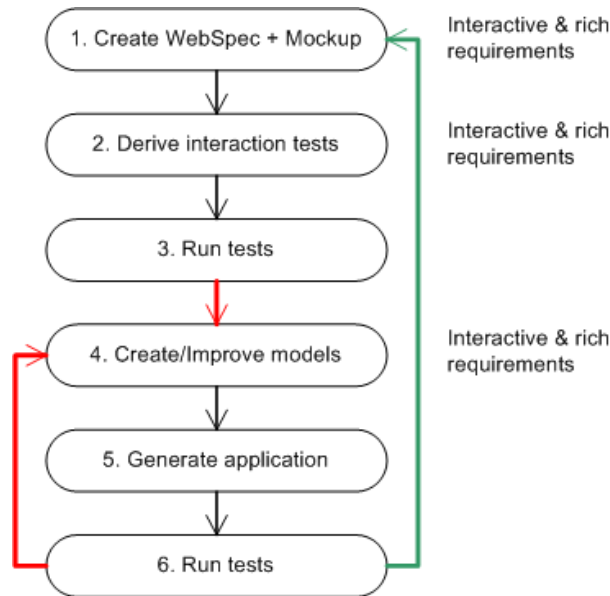


Fig. A.1. Extensions to WebSpec's core language to express rich behaviour

The content of this chapter is a submitted paper to the *Journal of Requirements Engineering*. This journal provides a focus for the dissemination of new results about the elicitation, representation and validation of requirements of software intensive information systems or applications.

WebSpec: a Visual Language for Specifying Interaction and Navigation Requirements in Web Applications

Esteban Robles Luna¹, Gustavo Rossi^{1,2}, Irene Garrigós³

¹*LIFIA, Facultad de Informática, UNLP, La Plata, Argentina*
{esteban.robles, gustavo}@lifia.info.unlp.edu.ar

²*Also at Conicet*

³*Lucentia Research Group, DLSI, University of Alicante, Spain*
igarrigos@dlsi.ua.es

Abstract. Web application development is a complex and time consuming process that involves different stakeholders (ranging from customers to developers); these applications have some unique characteristics like navigational access to information, sophisticated interaction features, etc. However, there have been few proposals to represent those requirements that are specific to Web applications. Consequently, validation of requirements (e.g. in acceptance tests) is usually informal, and as a result troublesome. To overcome these problems, we present WebSpec, a domain specific language for specifying the most relevant and characteristic requirements of Web applications: those involving interaction and navigation. We describe WebSpec diagrams, discussing their abstraction and expressive power. With a simple though realistic example we show how we have used WebSpec in the context of an agile Web development approach discussing several issues such as automatic test generation, management of changes in requirements, and improving the understanding of the diagrams through application simulation.

1 Introduction

Several studies [1, 2] in industrial cases have shown the importance of requirements in Web application development. Unfortunately, in this kind of applications, requirements are generally described in informal documents (e.g. use cases [3]) shared by the different stakeholders of the project, which are very poor to express the particularities of the Web (e.g. their interactive and navigation-driven nature). The fact that development teams are usually multidisciplinary (including customers, visual designers, developers, QA staff, etc) and that Web application requirements change very fast (e.g. as the result of early users' feedback), make things even harder.

The fast evolution of Web applications poses additional constraints to allow continuous and timely application testing against the requirement specification [2]. In this context, capturing and modeling requirements should be efficient enough to accomplish the time constraint. Moreover, requirement artifacts have to be easily understood to be validated by stakeholders prior to the development, in order to avoid future wastes of time. Moreover, as in “ordinary” software, during the development process the application has to be checked in order to validate that new requirements have been correctly implemented without “breaking” previous ones.

In the context of model driven Web engineering approaches [4, 5, 6, 7, 8] the aforementioned concerns have not been generally taken into account [9]. As a consequence, Web applications developed with these methodologies might suffer well-known problems such as outdated requirements, unfeasibility to check that the application fulfils the requirements and it might be difficult to handle fast evolution.

Existing languages to model Web requirements e.g. user interaction diagrams [4], extended use cases [10], etc. are useful to capture important aspects of Web applications like navigation or interaction issues; however they are at most used to document the application [3] or in some cases to help deriving the first version of the domain or navigation models [11, 12] and generally do not consider either evolution or validation (see Sect. 6 for further details).

To tackle these problems we have developed WebSpec, a multi purpose domain specific language used to capture navigation, interaction and UI (User Interface) features in Web applications. To improve the requirements capture, WebSpec is used in conjunction with mockups

(sketches of UI) to provide realistic UI simulations. Also, to allow fast requirements' validation in the final application, the associated WebSpec tool automatically derives a set of interaction tests. Finally, WebSpec enforces change management support, which could be used to improve the development cycle by automating structural changes in the application. Since WebSpec diagrams are intuitive and simple, they are suitable to drive discussions between stakeholders. The WebSpec language supports a powerful composition model, improving their scalability for complex applications. Finally, the WebSpec metamodel is open-ended, therefore allowing to broaden the scope of features that can be represented in a diagram (as an example we have extended the metamodel to incorporate rich interactions).

In this paper we present the WebSpec formalism, describing its components and the role they play in the development process; we emphasize on its novel features and show how to:

- Simulate the application using WebSpec and mockups to improve their understanding between the different stakeholders and reduce elicitation times.
- Derive tests from WebSpec diagrams to reduce requirement validation times.
- Capture requirement changes and use them to semi/automatically upgrade the application and maintain quality standards.

Additionally, we present a tool we have developed to create and manage WebSpec diagrams and describe in more details how WebSpec's features have been implemented.

The rest of the paper is structured as follows: in Section 2 we present WebSpec, its concepts and syntax. In Section 3 we show how WebSpec is used in different activities in the development cycle by improving requirement's elicitation, helping to automatically validate the requirements and manage their changes. Section 4 shows the WebSpec Eclipse plugin covering the implementation of its features. In Section 5 we present a case of study showing how WebSpec has been used for the development of a Web application for the post-graduate area of the College of Medicine in the University of La Plata. Section 6 presents related work and finally in Section 7 we conclude and present further work.

2 WebSpec: a DSL to capture interactive Web requirements

Web applications tend to change fast and it is hard for development teams to adapt to those changes easily. As part of the solution, the proliferation of agile practices [13] has improved the overall process as they have a continuous feedback from the different stakeholders. In these practices, requirements are captured informally [3] and as a consequence checking if they have been correctly implemented is sometimes impossible [1, 2]. Usually, development teams add manually created tests not only to check software artifacts but also to guide design decisions like in TDD (Test Driven Development) [14]. When the application evolves and the number of implemented requirements grows, tests are particularly necessary in order to verify that every unchanged requirement remains implemented in the application (known in the literature as *regression testing* [15]).

In order to capture Web requirements, researchers have borrowed use cases and user stories [13] from the software engineering field and try to use/adapt them in the Web engineering field (e.g. extended use cases). These artifacts allow describing the requirements in semi-structural/natural language making them flexible and appropriated to interact with customers. However, they do not help to describe UI aspects which are essential in Web applications, and as a consequence the validation of their correct realization in the application is performed manually. Moreover, validation is only performed over the last set of implemented requirements (due to the fact that the time spent on validating every requirement grows (in the best case) linearly with respect to the number of requirements implemented) and thus those side effects that affect previous requirements are not detected until a user finds a bug in the application.

On the other hand, there are more formal languages [16, 4] that help to specify interactive requirements more precisely, making easier for the development team to implement them since they usually provide some kind of automatic derivation of the basic application's structure (e.g. the topology of pages and the links between them). However, they usually do not provide automatic derivation of tests and those that are related with a specific model driven Web engineering approach (MDWE) [17] tend to be tightly coupled to the other modeling constructs of the approach. To make matters worse, many times they are too abstract or complex to be used or understood by customers and therefore unrealistic to be used in real life projects.

To tackle these problems, but preserving the advantages of the aforementioned languages, we have developed WebSpec. WebSpec is a visual language which has support for simulation (Sect.

3.2) helping customers visualize the requirement prior to its implementation. Requirement validation is done automatically (Sect. 3.3) by running a test suite obtained from the requirements specification, which is independent of the implementation technology used as it is based on Web browsers and not in the technology used to develop the application. As any formal language it also provides derivation of some parts of the application (Sect. 3.4) to a particular technology (GWT [18], Seaside [19], etc) all integrated in its supporting tool, the WebSpec Eclipse plugin (Sect. 4).

WebSpec is a visual domain specific language ([20]) that allows specifying navigation, interaction and UI Web requirements. The main artifact for specifying requirements is the WebSpec diagram (Sect. 2.1) which can contain *interactions* (Sect. 2.2), *navigations* and *rich behaviors* (Sect. 2.3). As one of the main motivations of the language is automatic test derivation, we borrow the idea of generator [21] to specify properties that the application must satisfy. For example any of the following properties: “the price of a product must be a positive number” or “a valid username is a string of length between 8 and 16 composed of letters and numbers” can be specified using a generator. A generator (Sect. 2.4) provides a simple and reusable way to describe a data set (by extension or comprehension); it can be interpreted as a function that returns a random element of the specified set. For example a string generator configured with minimum length of 8 and maximum length of 16 could be used to obtain valid usernames for the aforementioned case (e.g. “administrator”). Finally, WebSpec diagrams can be composed (Sect 2.5) to cope with complexity and at the same time to allow reuse of requirements.

WebSpec is formally defined in the metamodel shown in Fig. 1. For the sake of conciseness we avoid the Expression and Widget hierarchies but the reader could find more information in Appendix A. A diagram (instance of the class Diagram) comprises Interactions and Transitions (either Navigation or RichBehavior) instances. An Interaction instance knows its name, forward transitions and its associated interface mockup. A Transition knows its source and target Interaction, its precondition and the sequence of Action instances that triggers them. Finally, an Interaction knows its root widget Container which can contain many AbstractWidget (Widget or Container) instances. Each widget can also be associated with its representation in the mockup using its location attribute.

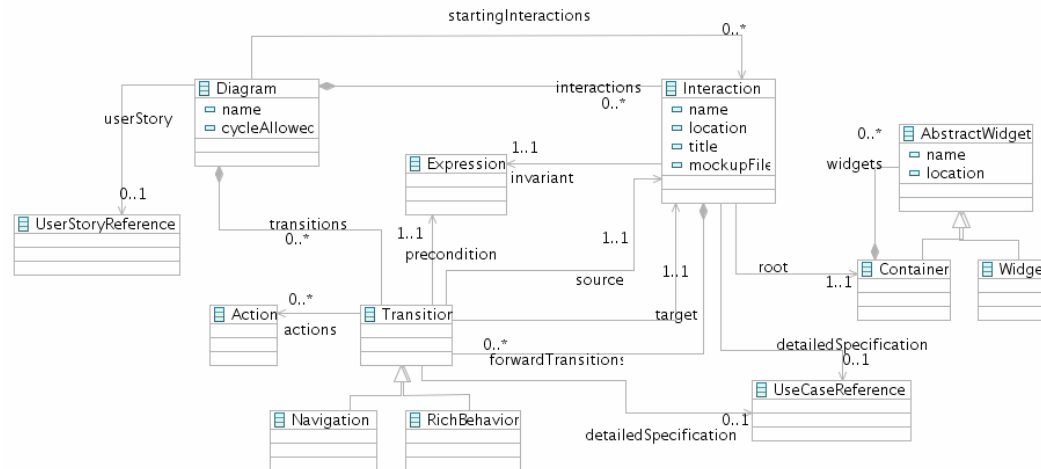


Figure 1. WebSpec simplified metamodel

In the following subsections we will introduce the aforementioned concepts using an example of an e-commerce application. The language will be described with a simple user story: “As a customer, I would like to search products by name and see its details”.

2.1 WebSpec Diagrams

A WebSpec diagram defines a set of scenarios that the Web application must satisfy. It can contain two main elements: *interactions* and *transitions* (which can be in turn *navigations* or *rich behaviors*). *Interactions* represent points where the user can interact with the application and *transitions* represent a movement from one point of interaction to another. Therefore, a WebSpec diagram could be seen as a graph where *interactions* are the nodes of the graph and *transitions* represent the edges. A scenario is represented by a sequence of *interactions* and *transitions*, e.g. <interaction1, navigation1, interaction2, rich1, interaction3> that defines a possible path of interactions between the user and the Web application.

Fig. 2 shows a WebSpec diagram for our exemplar user story. The diagram is constructed iteratively between the customer and the analyst by having several meetings. Since the use of WebSpec is not tight to any particular development process, we can use the techniques which are common in unified development approaches or traditional customers meetings typical of agile development approaches to build them. Their construction could be improved by using mockups and simulating the application (Sect. 3.2); however, we expect that with some training the customer would be able to solely build a diagram. The diagram of Fig. 2 defines the navigation paths that the user can follow from the home page to the search results page and then to the details of the products. Also, the user is able to go back to the search results page from the detail of the product or go back to the home page.

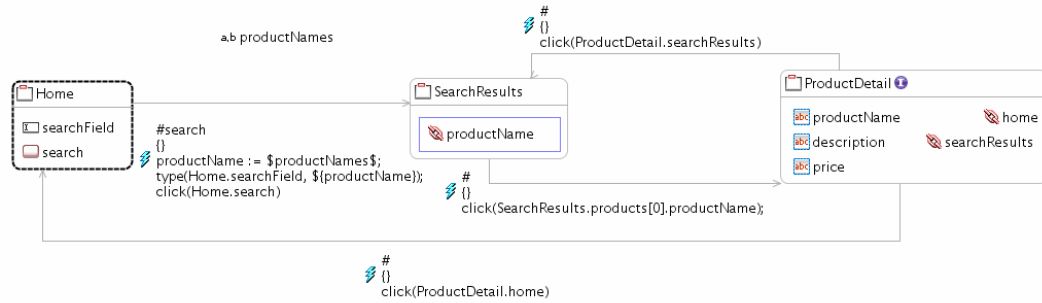


Figure 2. Webspec diagram of the *Search by name* scenario

The set of scenarios that the diagram specifies is obtained by traversing the diagram using the DFS algorithm [22]. The algorithm starts from a set of special nodes called “starting” nodes (Sect. 2.2) and follows the edges (transitions) of the graph (diagram). Typically, one or more diagrams could be related with the same user story to specify concrete scenarios that the Web application must satisfy. In the following sub-sections we elaborate the contents of the diagram.

2.2 Interactions

An *interaction* represents a point where the user can interact with the application by using its interface objects (widgets). Formally, they represent the state of a Web page either when it is loaded when the user navigates to it or when it has changed as a consequence of a rich behavior (Sect. 2.3). *Interactions* have a name (unique per diagram) and may have widgets such as: labels, list boxes, buttons, radio buttons, check boxes and panels. Labels define the content (information) shown by an *interaction*. There are two types of widgets that allow defining widgets composition: ListPanel and Panel. A ListPanel represents a repetition of the elements that it contains and the Panel defines a simple placeholder that can contain any simple or composed widget. *Interactions* are graphically represented with a rounded rectangle (Fig. 3) which contains the *interaction's* name and widgets. A WebSpec diagram must have at least one starting *interaction* represented with dashed lines.

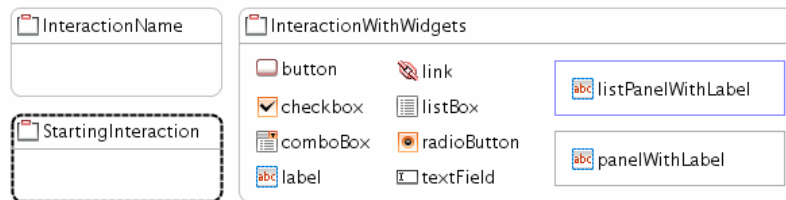


Figure 3. WebSpec's interaction

To specify which properties must be satisfied by the application we use invariants (Boolean expressions) on the diagrams' *interactions*. Every *interaction* (either implicitly or explicitly) defines an invariant that specifies which properties must be satisfied in the set of scenarios specified by the diagram (in case that we do not define one explicitly, it is implicitly assumed that the invariant is *true*). Boolean expressions may refer to the following elements:

- Widgets properties: Any property of a widget that is contained in the *interaction*. For example, *ProductDetail.productName.text* refers to the text value of the productName widget and is valid if it is contained in the invariant of the ProductDetail *interaction*.

- Variables: When we need to refer to a value or the property of a previous *interaction* in the scenario, we need to store them in variables, e.g. *productName := "ipod"* or *productName := ProductDetail.productName.text*. We refer to the value of the variables using the following syntax *#{variableName}* inside invariants.
- Generators: As we will show in Sect. 2.4, generators can be referenced using the following syntax *\$generatorName\$*, e.g. *productName := \$prods\$*.
- Composed expressions: It is possible to compose expressions using and (&&), or (||), implications (->) and negations (!). Please, refer to the Appendix A for the complete grammar.

As an example, the ProductDetail *interaction* of Fig. 2 defines an invariant (marked with the I icon near the *interaction's* name): *ProductDetail.productName.text = #{productName}* that states that the text of the productName label must be equal to the value of the productName variable. To improve the clarity of the diagram, we avoid showing them directly as the expressions could be quite complex. Instead, interactions are marked with an icon and the expression could be edited by changing the interaction's property in our Eclipse tool (Sect. 4).



Figure 4. Product details mockup created with Balsamiq

To improve the understanding of the diagrams by the different stakeholders, we can associate *interactions* with mockups and WebSpec widgets with their concrete UI elements in the mockup. Using this association, we can switch between the specifications in WebSpec with an exemplar UI that will help to understand the requirements. Mockups can be created with tools such as Balsamiq [23], Axure [24] or plain HTML and can be developed by, or with the participation of customers. For example in Fig 4, we show a mockup of the product details page created with Balsamiq. The mockup shows the information that must be presented on that page: the product name, its description, price and the links to the home and search results. Fig. 5 shows a simple association between the mockup of Fig. 4 with its corresponding *interaction* and widgets of Fig. 2.



Figure 5. Association between a mockup and its corresponding interaction

2.3 Specifying the application's behavior

Usually, the behavior of Web applications is exercised either by navigating from one page to another or by local (interface) changes that may not involve navigation to a new page. These

behaviors are perceived by the user by changes in its browsing history or in the UI respectively; therefore we will call them Interactive behaviors (Sect. 2.3.1). On the other hand, there are behaviors that are not directly perceived by the user and are triggered as a consequence of navigating from one page to the other. Examples of such behaviors are: sending an email, charging a credit card, or even making a search in Google using the Google’s API; these can be informally specified in WebSpec using either notes or by associating WebSpec’s elements with use cases (Sect. 2.3.2).

2.3.1 Interactive behaviors

When the user navigates from one page to another, a new element in its browsing history is added allowing him to go back to the previous page. During requirements elicitation these elements are easily identified by the analyst in the customers’ vocabulary when they say “In this page, I would like to allow users to go back to the previous page”.

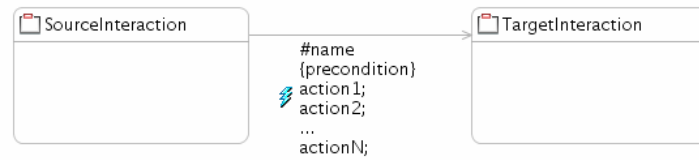


Figure 6. WebSpec’s navigation

In WebSpec, a *navigation* is graphically represented (Fig. 6) with grey arrows while its name, precondition and triggering actions are displayed as labels over them. In particular, its name appears with a prefix of the character ‘#’, the precondition between {} and the actions in the following lines. We must remark that the idea behind the transitions’ actions (either *navigations* or *rich behaviors*) is that the execution of them produces the transition between *interactions* and not in the other way. A *transition* should be understood like: “if the precondition holds **and** the user executes the sequence of actions **then** the application should transit to the target *interaction*”.

A *navigation* from one *interaction* to another can be activated if its precondition holds, by executing the sequence of triggering actions such as: clicking a button, adding some text in a text field, etc. As well as invariants, preconditions can reference variables declared previously in the diagram. Actions are written according to the following syntax: `var := expr | actionName(arg1,... argn)` (a complete BNF [25] grammar can be found in the Appendix A).

Traditional hyperlink navigation is represented with no precondition (indeed, an always true precondition) and with only one action *click* (a link widget), as illustrated with the ProductDetail to Home navigation in Fig 2. An example of a more complex sequence of actions is the *search navigation* (Fig. 2):

```

(1) productName := $productNames$;
(2) type(Home.searchField, ${productName});
(3) click(Home.search);

```

The first sentence assigns the data generated by the productNames generator (denoted between \$) in the productName variable (for later use). In the second sentence the content of the productName variable is typed in the searchField text field, and finally in the third sentence the search button is clicked.

On the other hand, the application may change its UI state as a consequence of some actions performed by the user (e.g. on some interface widgets). For example, when the mouse is “on” a widget, some additional information might pop-up, or while entering text in a field, the text might be auto-completed. These “local” changes are common in the so-called rich Internet applications [26] and it is nowadays usual that customers pose requirements of this type, either explicitly (“I want an auto-complete feature in this field”), or implicitly (“I want that information appears as in Amazon.com”). These “rich” behaviors are being increasingly used not only in Web 2.0 applications but also in traditional, e.g. e-commerce, ones.

In a Web application, a rich behavior is perceived by a local change in the UI of the Web application and it **does not** add a new element in the browsing history. To specify a *rich behavior* in Webspec, we use a red dashed arrow (Fig 7) though it has the same properties that a *navigation* has (name, precondition and actions).

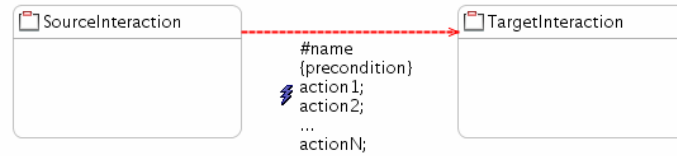


Figure 7. Rich behavior specification in WebSpec

Fig. 8 is an extension of Fig. 2 which shows a specification for the Hover detail pattern [27] in the search result list. This pattern gives more information about an item when the user puts its mouse over an item. In this case, a detail of the product is shown (SearchResultsProductHover interaction) and allows the user to navigate to the product details page. Notice that from an *interaction* reached as a consequence of a rich behavior we can also have *navigations* and *rich behaviors* to other *interactions* (SearchResultsProductHover *interaction* to ProductDetail *interaction*).

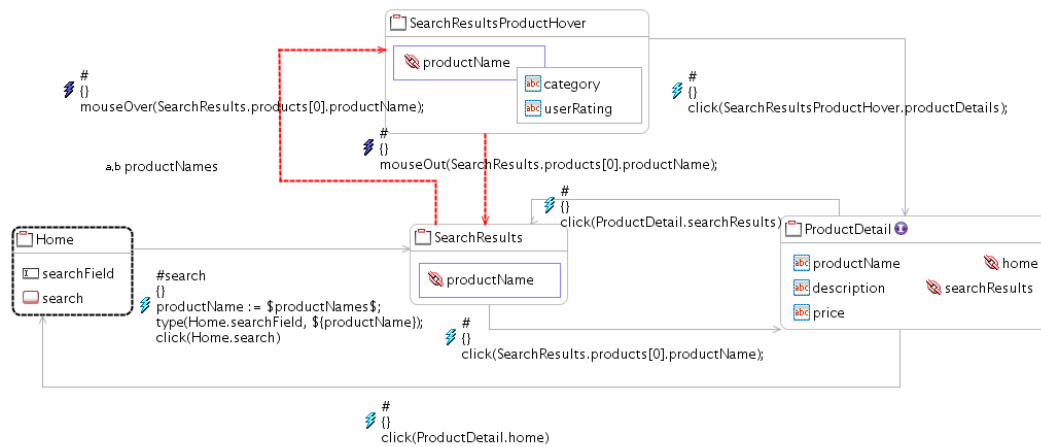


Figure 8. Hover detail in *SearchResults* interaction

2.3.2 Dealing with “non interactive” behaviors

Most Web application requirements are related with interactive aspects that can be specified using invariants and actions. However as said before, there are some scenarios that may have important “hidden” behaviors (not perceived directly by the user from an interaction point of view) and that are important to be specified.

To capture this kind of requirements, Webspec can be combined with two different artifacts (depending on the needed level of detail) for specifying hidden behavior. If we need to specify simple functionality that does not require complex business rules we can use informal notes that can be added to the diagram and/or linked to *interactions* or *transitions*. Notes provide an easy way of specifying some details that will not be perceived from a user point of view. Fig. 9 adds a note to the search *navigation* to explain that the search operation should be implemented by integrating with Google Search.

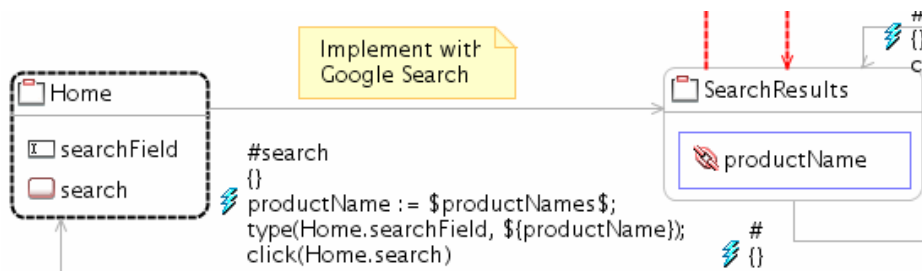


Figure 9. Note explaining *Search* implementation details

On the other hand there are some complex cases, such as Web service calls, credit card transactions, etc, that can not be detailed using notes. We have identified the following categories:

- Complex integrations between Web (or other kind of) applications are usually difficult to achieve, and generally involve details such as APIs or other contracts, format of exchange data, etc. In these cases it is better to use detailed documents about these requirements.
- Low level technical details such as the information that needs to be stored in log files as part of a business process of the application. This information is generally stored in files on the server and therefore does not show up during user interaction.
- Any application's behavior that is not perceived from a UI point of view such as generating a PDF report with statistical data about the user's activity.

In all these cases, WebSpec allows linking *interactions* and *transitions* with use cases for a more complete description of the requirement (see the association between Interaction and Transition classes with UseCaseReference in the metamodel of Fig 1).

2.4 Specifying Properties with Generators

With WebSpec it is possible to specify general and concrete application properties. A concrete property is specified with one or more scenarios that use constant values in Actions (e.g. *type(Login.username, "admin")*) and/or Invariants (*Home.messages.text = "Welcome admin"*). On the other hand, sometimes it is necessary to specify more complex properties like "an error must be shown if the user tries to add a comment larger than 150 characters to a product" for any comment (any string of at most 150 characters).

To specify general properties, we can create the diagram with concrete values and then abstract them using generators. Generators are necessary to map abstract scenarios (those without concrete values) to concrete scenario instances (with the corresponding data distribution). This mapping is used during test generation (Sect 3.3) and simulation (Sect 3.2). A generator helps to define which are the valid data sets for the different scenarios and help the development team (as it is a formal definition of a data set) to implement each scenario accordingly to the expected logic.

Following the idea of QuickCheck [21], we extract the data used for specifying interaction requirements into generators. If a property in a WebSpec diagram holds, then it must hold for any element that could be generated by a generator. A generator is a function that can be called from transition actions (e.g. *\$productName\$*) and generates data. For example, Fig. 2 has one generator that generates product names for searching purposes. A generator can also be seen as a definition of a set by comprehension; for example the generator *usernames* = all the strings of length between 8 and 16 that contains letters or numbers (*{aaaaaaaa, aaaaaaab, ... }*).

With the aim of specifying different types of requirements, WebSpec provides a variety of generators based on the ones QuickCheck already provides; though adding a new generator is not a hard task. Next, we detail the generators actually provided in WebSpec:

- One of many strings: The user can specify a set of strings and the generator chooses one with uniform distribution probability. For example, if the generator is configured with: "Home", "Ipod", "Smartphone", the generator could generate the string "Ipod".
- One of many numbers: Similar to one of many strings. For example the user can configure the generator with 4, 5, 8, 10.5, 2, -1 and the generator could generate the number 8.
- Uniform distribution of numbers: The user configures minimum and maximum values and the generator picks a number in the continuous interval with equal probability. For example for the interval [3.76, 15] the generator could generate the number 8.7.
- Random string: The user configures the minimum and maximum length of the string and the generator generates a random string with a length in the specified interval. For example for the interval [2, 10] the generator could generate "agfasg".
- One of many arrays: The user configures a set of arrays and the generator chooses one with equal probability of being chosen. We use arrays when there are interdependencies between data. For example, the arrays of valid users that have username and password: [admin, admin], [john, johnpass], [root, superuser]; thus the generator could generate the array: [admin, admin].

Each of these generators has a visual representation shown in Fig. 10.

a,b	One of strings	[0,1]	Uniform distribution of number
1,2	One of numbers	a*	Random string
n.n	One of arrays		

Figure 10. Different types of WebSpec's generators

2.5 Diagrams' composition

When the application grows, new requirements may refer to previous described (and perhaps finer grained) requirements. Let us assume that we have the following requirements expressed as user stories: "As an administrator, I would like to search for users by email in order to ban them" and "As an administrator, I would like to check the user's activity searching them by email". Both refer to some functionality of the administrator regarding actions they would like to perform: search by email, banning and check user's activity. Fig 11a and Fig 11b shows the WebSpec diagrams corresponding to each requirement.

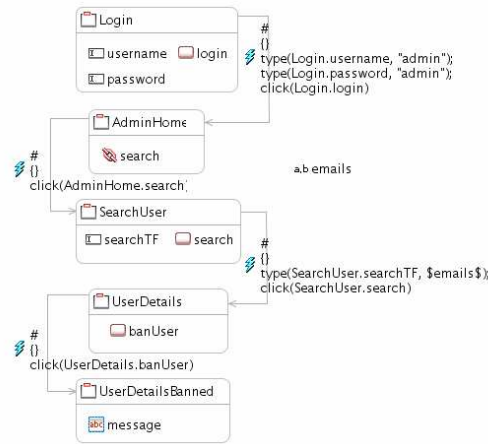


Figure 11a. Ban user diagram

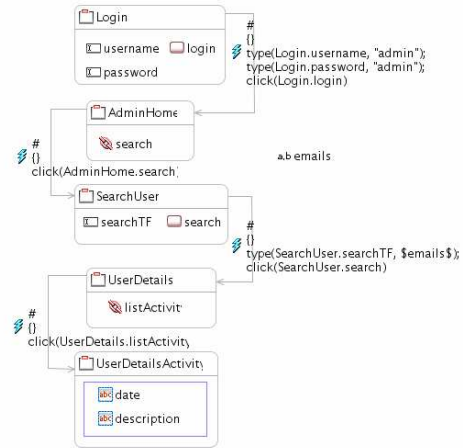


Figure 11b. Check User's activity diagram

Notice that both diagrams have a common sequence of *interactions* and *transitions* that sets the preconditions to be able to express the requirement. In this case the sub-path: Login -> AdminHome -> SearchUser is common to both diagrams and its main intent is to login with an admin user and search for a user in the system. The *interactions* and *navigations* that follow this sub-path are the ones that actually express the requirement.

To improve the understanding and scalability via composition, we define the concept of operation as a path that can be composed in other diagrams or operations. Fig 12 shows the definition of the LoginAsAdminAndSearchForUser operation.

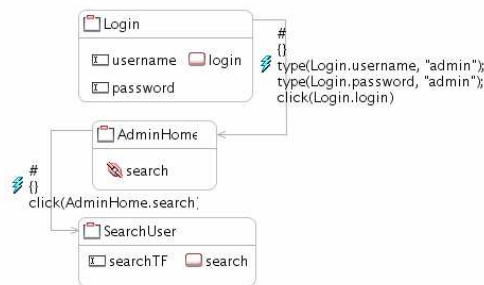


Figure 12. LoginAsAdminAndSearchForUser operation

As a consequence, the diagrams of Fig. 11a and Fig. 11b can be written in a more shortly way as shown in Fig. 13a and Fig. 13b. These diagrams are the composition between the LoginAsAdminAndSearchForUser operation and the sub-paths of Fig. 11a and Fig. 11b.

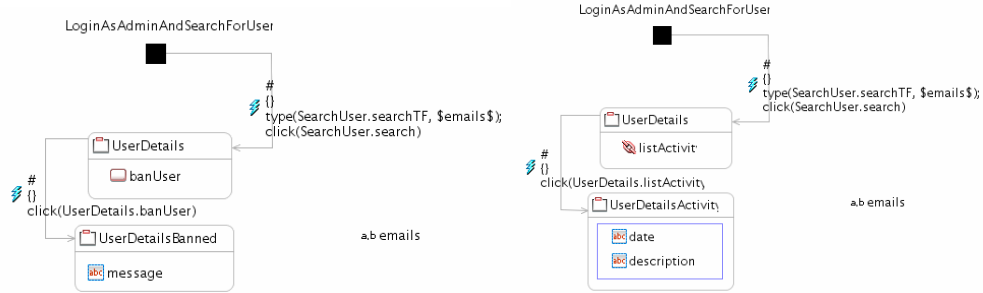


Figure 13a. Refactored *Ban user* diagram Figure 13b. Refactored *Check User's activity* diagram

2.6 WebSpec guidelines

When using WebSpec for Web requirements specification, diagrams tend to grow with the time, thus hindering comprehension; as a consequence we have written several simple guidelines to be taken into account during the development process:

- **Similar interactions:** When two or more diagrams have an *interaction* with the same name, we will assume that two *interactions* denote the same point of interaction. In this way, when a stakeholder looks at two different diagrams and they see *interactions* with the same name, they will know that they denote the same point of interaction improving comprehension.
- **Explicit specification:** If a widget *w* is present in the *interaction* *A* of diagram *D1* and widget *w* is absent in the *interaction* *A* of diagram *D2* then it does not mean that the widget has been deleted. Indeed, it means that the widget is not meaningful for the specification.
- **User story/Use case association:** As the application evolves, the number of diagrams tends to grow quickly thus it is important to keep track of which user story gives origin to a diagram. This could be easily done by linking a diagram with its corresponding user story (see the association between Diagram and UserStoryReference in Fig. 1).

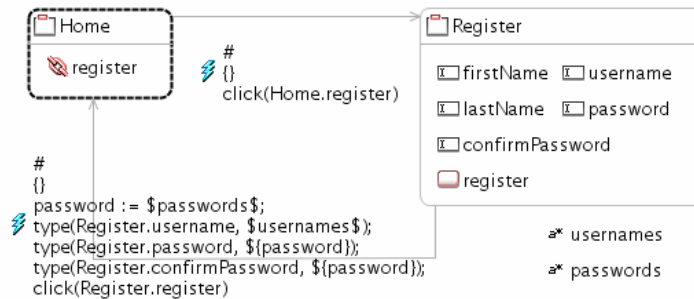


Figure 14. *Register* WebSpec's diagram

As an example, we have added a new diagram to the one in Fig 2 that specifies the register User story. Fig 14 shows the Register user story and it shows a Home *interaction* which has the same name to the one previously created in Fig 1. According to the first guideline they refer to the same point of interaction. Also, the two versions of the Home *interaction* have different widgets inside; a search button and a searchField field in one case and a register link in the second one. According to the second guideline, the absence of the search button in Fig 14 does not mean that the widget has been deleted. If we want to specify that the widget is not visible then, the widget has to be added to the *interaction* and the invariant must contain an expression like: *!Home.search.visible*

3 WebSpec in use

In the previous sections we have presented the language and the way in which we specify interactive requirements in Web applications; in this section we explain how Webspec is used in the development cycle. As an introduction we detail how a diagram that has cycles and specifies infinite scenarios is used in practice (Sect. 3.1). Next we show WebSpec's features such as simulation of the application (Sect. 3.2), requirement validation (Sect. 3.3) and requirement changes (Sect. 3.4).

3.1 Bounding infinite scenarios

As the diagram of Fig. 2, WebSpec's diagrams may specify an infinite set of scenarios when they have cycles. For example, the diagram of Fig. 2 has a short cycle between SearchResults and ProductDetail *interactions*. So if the diagram specifies such infinite scenarios how are we going to simulate the application or validate that the requirements are correctly implemented? In both cases, we have adopted a pragmatic approach; as the scenarios are infinite and either the simulation or validation would not end, we prune those "infinite" paths according to a maximum occurrence (constant) of an *interaction*. Therefore, a scenario can either end on an *interaction* with no transition or when the number of occurrences of an *interaction* reaches a maximum number set per diagram.

To have a better idea of what pruning means in this context, let us look at our example of Fig. 2; we will allow a path to contain as much as two occurrences of the same *interaction* in the path. We have chosen that number because we would like to have concrete scenarios where the user goes through the same *interaction* more than once. In order to compute the scenarios we start transversing the diagram starting from the starting *interactions* and following the diagram using the DFS algorithm. Therefore, the algorithm starts from the Home *interaction* and follows the SearchResults and ProductDetail *interactions*. The paths shown below are the ones computed by the algorithm. In the first case, the algorithm stops because either if we add Home or SearchResults *interactions*, we will violate the maximum occurrences of 2 elements. The same applies for the 2nd path. The paths computed are shown next:

```
Home -> SearchResults -> ProductDetail -> Home -> SearchResults -> ProductDetail
Home -> SearchResults -> ProductDetail -> SearchResults -> ProductDetail -> Home
```

If the diagram has cycles, WebSpec forces to define a maximum number of occurrences for the same *interaction*. The number to be set really depends on the requirement we are specifying; for instance if we are specifying the add to cart requirement (which is an important requirement of an e-commerce application), we may allow 10 occurrences of the same *interaction* when trying to validate them on the final application just to be a bit more sure that the application behaves as expected.

3.2 Improving team understanding with WebSpec and Mockups

With the aim of improving the requirement elicitation phase, WebSpec diagrams allow the simulation of the application under development. Simulation is important to bridge the gap between the understanding of customers and analysts about requirements, thus helping to get real feedback from them. Usually, requirement artifacts [28] require some level of knowledge from customers to be fully understood, causing understanding problems during elicitation that are handled lately when the application is under active development.

Understanding a WebSpec diagram may not be a simple task; it takes time and requires the knowledge of WebSpec's concepts, e.g. variables and *interactions*. To ameliorate this scenario WebSpec provides some interesting features such as mockup associations (Sect 2.2) and invariants specification which allow simulating the application in a rather rigorous way to improve their understanding between stakeholders during elicitation. Our simulation basically opens a Web browser with the developed mockups and show descriptions and performs actions that show how a hypothetical user would interact with the application. It is rigorous, because differently from the simulation provided by tools such as Balsamiq [23], we not only show transitions between the pages but also execute real actions and provide descriptions of what would be the real output of the application, directly over mockups. These descriptions are generated automatically from the WebSpec diagrams, and they are easy to understand because they are written in natural language. In this way, from every WebSpec diagram, a set of simulations is automatically generated which can be used at any time by customers to understand the meaning of the diagram and suggest changes or improvements to the analyst.

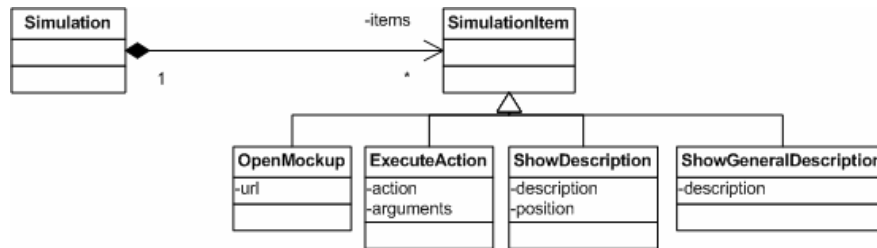


Figure 15. Simulation metamodel

The interaction between the development team and the customer starts by specifying a diagram and usually involves the creation of some mockups. During this process each interaction and its widgets are associated with their corresponding elements in the mockup as shown in Sect. 2.2. Afterwards an automatic transformation is applied over the diagram obtaining a set of scenarios. Then, a simulation is derived from each scenario and captured as instances of the metamodel shown in Fig. 15.

A simulation contains several steps (items) that must be executed (on the Web browser) to simulate the scenario. Those items are the following:

- **OpenMockup:** it opens the mockup in the specified URL.
- **ExecuteAction:** Executes the action over an already opened mockup with some arguments.
- **ShowDescription:** Shows the description at a specific position.
- **ShowGeneralDescription:** Shows the description covering the full page.

Each simulation is created following the sequence of *interactions* and *transitions* of a concrete scenario. Next, we show a simplified version of the transformation algorithm written in natural language:

```

(01) Create a simulation S for the scenario C.
(02) for each element E in the scenario C {
(03)   if (E is an Interaction) {
(04)     Open the mockup M associated with E.
(05)     Show a description that must hold according to E's invariant.
(06)   } else {
(07)     Show a description that must hold according to E's precondition.
(08)     for each action A in the transition E {
(09)       Simulate the action A over the mockup M.
(10)     }
(11)   }
(12) }
  
```

Line 1 creates the simulation model; for every item (*interaction* or *transition*) in the path (2): if it is an *interaction* (3) we show its associated mockup (4) and show the predicate of its invariant to describe which properties must hold (e.g. “The label should have the value ‘John’”) (5); if the item is a *transition*, we show the precondition (7) and for every action we simulate it (08-10).

As an example, let us consider the scenario Home -> SearchResults -> ProductDetail -> Home -> SearchResults -> ProductDetail. As the model generated by the algorithm includes 16 instances of SimulationItem we show next a text representation of the same instances so that they can be easily understood.

```

(01) Open Home's mockup.
(02) Execute action type on searchField with value "Ipod".
(03) Execute action click on search.
(04) Open SearchResults's mockup.
(05) Execute action click on productName.
(06) Open ProductDetail's mockup.
(07) Show description at productName with value "The value should be 'Ipod'".
(08) Execute action click on home.
(09) Open Home's mockup.
(10) Execute action type on searchField with value "book".
(11) Execute action click on search.
(12) Open SearchResults's mockup.
(13) Execute action click on productName.
(14) Open ProductDetail's mockup.
(15) Show description at productName with value "The value should be 'book'".
(16) Execute action click on home.
  
```

After an instance of the Simulation metamodel is created, the application can be simulated inside a Web browser by opening mockups in the browser, showing descriptions and performing actions on it. In Sect. 4 we provide details of how this feature has been implemented in our Eclipse plugin.

3.3 Requirements validation

New requirements must be validated to guarantee their correct implementation while previous ones still work as intended. However, it is hard to perform this task efficiently, therefore keeping the requirements updated is extremely important.

A well known way of validating requirements consists in running automated tests (that express the requirements) over the application. If one of these tests fails, then a requirement is not satisfied by the application. In particular, interaction tests play an important role in industrial settings as they execute a set of actions in the same way a user would do on a real Web browser, thus their use is continuously growing [29]. As an example, in a recent work we have introduced the use of interaction tests in the WebTDD test/model-driven approach [30].

The test suite (a set of test cases) is built from a WebSpec diagram by creating a test for each scenario that the application must satisfy. To capture the basic concepts of tests, we have created a metamodel (Fig. 16) which is independent of the automated test technology used. The metamodel contains the Test and TestSuite classes that conceptualize a test and a set of tests. A Test has a sequence of actions: assertions on interface objects or actions performed by the user over the application. Both cases are covered by the TestItem hierarchy.

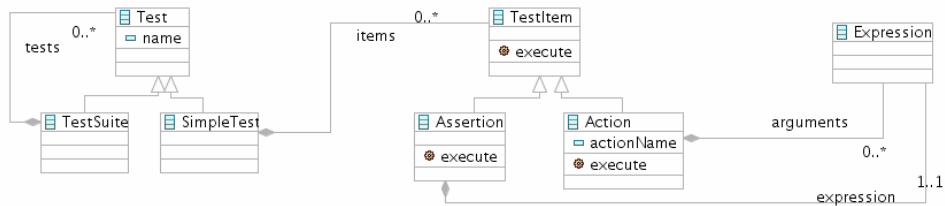


Figure 16. Test metamodel

To build the test suite, we transform each scenario into a SimpleTest (see Fig. 16) by executing the following simplified version of the algorithm. Similar to simulations, we use generators to generate data according to the specification when an expression references it. The TestSuite is obtained by simple composition (see the composition relationship in the metamodel of Fig. 16) of the previous SimpleTest instances.

```

(01) Create a test T for the scenario C.
(02) Add an item to open the URL of the application in T.
(03) for each element E in the scenario C {
(04)   if (E is an Interaction) {
(05)     Add an assertion that must hold according to E's invariant.
(06)   } else {
(07)     for each action A in the transition E {
(08)       Add an execution of the action A.
(09)     }
(10)   }
(11) }
  
```

The algorithm works as follows: line 1 creates the test model and line 2 generates the action to open the application. For each element in the path: if it is an *interaction* (4), we assert its invariant (5); if it is a *transition* (7) we execute the actions that allow us to navigate from one interaction to another (7-9).

To better illustrate these ideas, let us consider a specific scenario: Home -> SearchResults -> ProductDetail -> Home -> SearchResults -> ProductDetail. Applying the previous algorithm to the scenario produces a test model with 16 TestItem instances; we show a textual version of the model so that it can be better understood.

```

(01) Open the URL of the application.
(02) Execute action type on searchField with value "Ipod".
(03) Execute action click on search.
(04) Wait for the page to load.
(05) Execute action click on productName.
(06) Wait for the page to load.
(07) Assert that the widget productName has the value "Ipod".
(08) Execute action click on home.
(09) Wait for the page to load.
(10) Execute action type on searchField with value "book".
(11) Execute action click on search.
(12) Wait for the page to load.
(13) Execute action click on productName.
(14) Wait for the page to load.
(15) Assert that the widget productName has the value "book".
(16) Execute action click on home.
  
```

After an instance of the test metamodel is created, the application can be validated using a technology-dependent interaction test framework which operates on a Web browser. In Sect. 4 we provide further details about the implementation of test derivation in our Eclipse plugin.

As aforementioned, Web applications tend to change very fast, thus recording requirements changes is important to improve the development process. In the next subsection we show how requirement changes are captured (Sect. 3.4.1) and later used to ease the evolution of the application under development (Sect. 3.4.2).

3.4 Requirement changes

3.4.1 Capturing requirement changes

Capturing requirements changes is an important feature to predict their impact in the application. Though some mature requirement artifacts [3] provide extensions to support change management, in the Web engineering field this issue has been often ignored (see Sect. 6 for details).

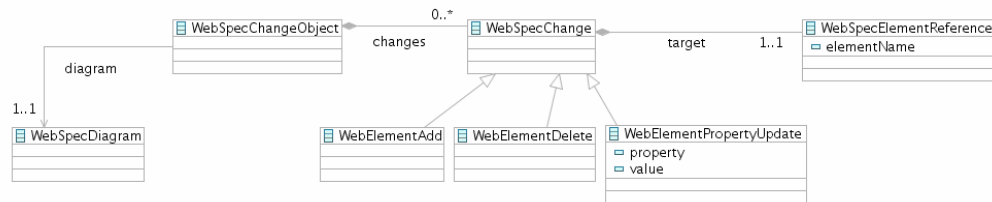


Figure 17. Change metamodel

In WebSpec, changes are recorded into change objects that group a set of changes. Change objects are created even in the initial stage (when a diagram is being created).

WebSpec diagrams can suffer different coarse grained changes, such as the addition or deletion of an *interaction* or *transition* element. These elements can be modified too, by the addition or deletion of widgets to an *interaction*, changes in invariants, etc. As for *transitions*, we can add or delete preconditions, change their source, target, or the actions that triggers them. All these types of possible changes have been represented in the metamodel of Fig. 17. When the user modifies the diagram, a change object is created and the sequence of changes is recorded as instances of this metamodel.

As an example, let us suppose that we add a Register *interaction* with its widgets and a link to it from the Home *interaction* (Fig. 18). The change in the diagram generates a new change object which has the following elements: the new *interaction* (Register), a new *navigation* (Home -> Register), a new link (register) in the Home *interaction* and set of widgets in the Register *interaction*.

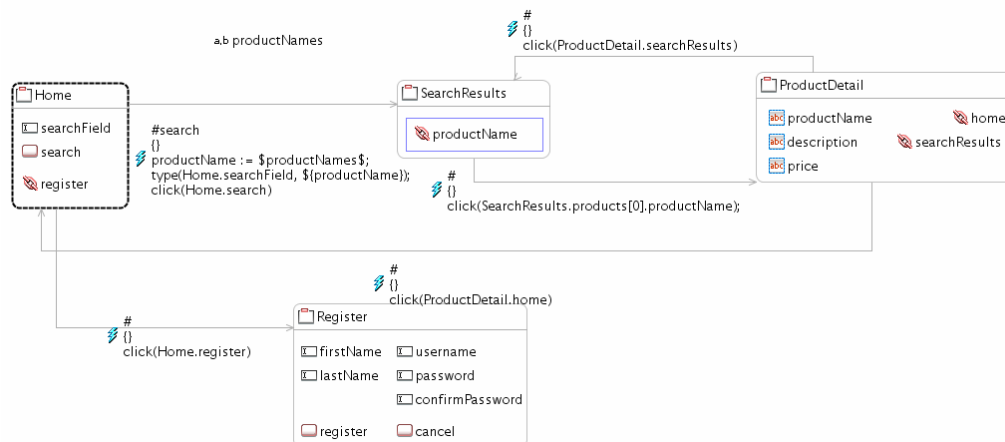


Figure 18. Adding a register page to our E-commerce application

In the following section we show how changes in the requirements help to upgrade the application under development to satisfy the new requirements.

3.4.2 Using requirement changes to ease application evolution

Though handling requirement changes serves for multiple useful purposes, we will focus on how to semi automatically upgrade the application using them. Since change objects represent changes at the WebSpec level (requirements), we decouple the process of upgrading the application by providing different effect handlers. An effect handler is a component responsible of mapping the changes in the diagrams to a concrete technology and storing the trace links between the WebSpec elements and the technology ones.

To keep the discussion at a conceptual level and show a concrete example, let us assume that the application under development is designed with classes and that we already have a version of the application. In Fig. 19 we show a class diagram of the classes involved in the UI model of our application before applying the change of Sect. 3.4.1.

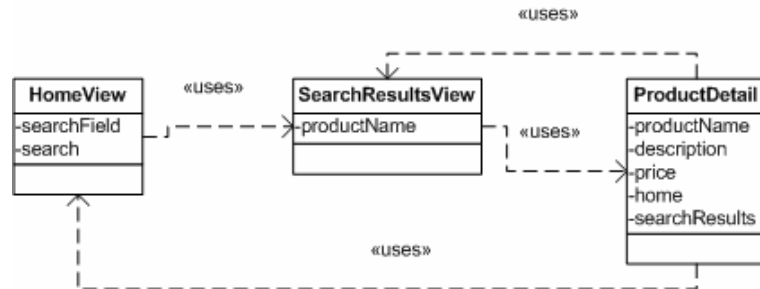


Figure 19. Class diagram of the UI model before applying the change

To upgrade the application after the changes, we need to define a mapping between the changes in WebSpec to the concrete implementation. In a class-based design, we have defined the following mapping:

- New Interaction: A new class is created.
- New Widget: A new instance variable and a creational method are created.
- Update Interaction/Widget name: The class or instance variable is renamed.
- Delete Interaction: The class is deleted if no other class references it.
- Delete Widget: The instance variable and the creational method are deleted if the instance variable has no references.

Using the previous mapping, we upgrade the UI model automatically and obtain a new UI model which is shown in Fig. 20. The RegisterView class is created with its corresponding instance variables. Also, the HomeView class is modified with a new instance variable register that contains the link to the Registerview. In the following section we show our implementation plugin and explain some details of its implementation.

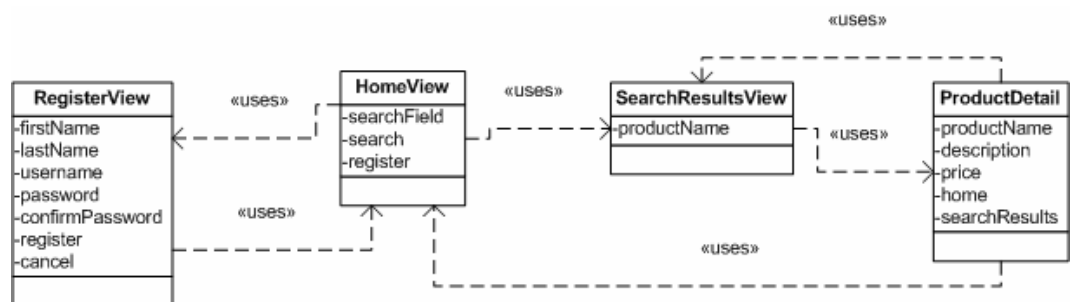


Figure 20. Upgraded version of the UI model after applying the change

4 Tool Support

A WebSpec tool has been implemented as an Eclipse plugin using EMF [31] and GMF [32] technologies; it is currently available as an open source project¹. The plugin supports the following features:

- **Creation of WebSpec diagrams:** a visual editor allows creating, modifying and updating diagrams. The properties of the elements can be modified by selecting each item and updating the property editors in the properties view.

¹ <http://code.google.com/p/webspec-language/>

- **Association with HTML mockups:** taking advantage of the Eclipse framework, HTML mockups are files inside the project. The editor allows selecting an *interaction* and associating it with the HTML file. Association between Webspec's widgets and HTML widgets is performed by editing the location property of Webspec's widget.
- **Simulation of the application:** Using the previous association, the plugin opens the mockups in the Web browser and show descriptions of what is the expected behavior. This feature has been implemented by extending the Selenium [33] communication mechanism and using a JQuery plugin [34] for showing the descriptions.
- **Selenium test derivation:** As previously shown, each diagram is transformed into a test model. Then, the plugin allows the translation of the test model into a Selenium test.
- **Change recording:** Using the EMF observer pattern [35], we hook on all changes that are performed in the diagram and the plugin creates a change model. The user of the plugin can set when should the plugin start recording changes and when not. When some changes are captured and the user stops recording, the change model is stored into a file for later use.
- **Generation/Update of GWT and Seaside UI classes:** Finally, using the previous stored change model, the UI model can be generated. Currently, the plugin allows the generation of GWT and Seaside classes and handles not only a first version of changes but also an incremental set of changes.

Fig. 21 shows a screenshot of the WebSpec's Eclipse plugin. In the following subsections we provide more details regarding the implementation of the aforementioned features in the plugin.

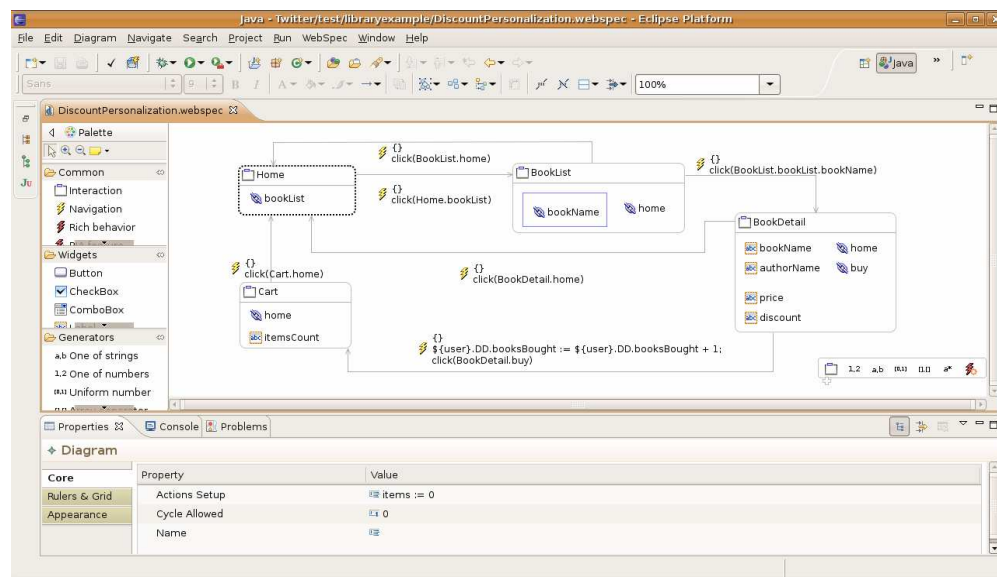


Figure 21. WebSpec's Eclipse plugin

4.1 Dealing with Simulation

The simulation feature comprises three elements: transformation between WebSpec and Simulation models, association with mockups and execution of the simulation. The transformation between WebSpec and the Simulation models has been implemented directly in Java as it was much simpler to deal with path computing algorithms than using QVT.

Mockups association has been easily implemented by taking advantage of the Eclipse environment. We add a new property for *interactions* and widgets and a file dialog to let the user choose the right HTML mockup.



Figure 22. WebSpec's simulation

The actual simulation aspect was more complex and required the extension to the Selenium framework. We used the existing communication mechanisms of Selenium to open the Web browser and execute actions. As shown in Fig. 22, we show descriptions over the mockups by using a JQuery plugin. To make it work, we had to extend the Selenium framework to load these libraries and actually show the descriptions when necessary. We must notice that the same mockup (which could be richer than the interaction since it has more widgets) could be used in multiple and different simulations. Our approach maintains the mockup as it is without removing any existing widgets because doing so will confuse the stakeholders about their presence or absence.

4.2 Requirements validation

The support for requirements validation has been implemented in a two phase process: transformation from WebSpec to Test models, and test derivation to a specific automated test technology. The transformation between the models has been implemented by taking advantage of the existing simulation architecture (the transformation module), since both transformations use path computing algorithms.

In order to perform test derivation to a specific technology, we transformed the test models into a plain text representation of the test. The plugin currently supports Selenium and we are working on the derivation to Webdriver [36]. As an example we show next the generated code for the Selenium framework for our example scenario:

```
(01) selenium.open("http://localhost:8080/index.html");
(02) selenium.type("id=searchField", "Ipod");
(03) selenium.click("id=search");
(04) selenium.waitForPageToLoad("30000");
(05) selenium.click("id=product0");
(06) selenium.waitForPageToLoad("30000");
(07) assertTrue(selenium.getText("id=productName").equals("Ipod"));
(08) selenium.click("id=home");
(09) selenium.waitForPageToLoad("30000");
(10) selenium.type("id=searchField", "book");
(11) selenium.click("id=search");
(12) selenium.waitForPageToLoad("30000");
(13) selenium.click("id=product0");
(14) selenium.waitForPageToLoad("30000");
(15) assertTrue(selenium.getText("id=productName").equals("book"));
(16) selenium.click("id=home");
```

Line 1 opens the application in the Web browser. Lines 02-04 search for Ipod product, lines 05-06 selects the first product and finally line 07 asserts that the selected product has the name Ipod. Lines 08-09 navigate to the Home page. Lines 10-12 search for book product, lines 13-14 select the first product and finally line 15 asserts that the selected product has the name book. Line 16 navigates to the Home page.

4.3 Requirement changes

When a diagram is modified, we record its changes and store them in change files. A change file is a serialization version of the change model presented in Section 3.4.1 in XML format. To capture the changes we use the observer pattern and incrementally build the change model; afterwards we serialize it into an XML file.

Changes are read and used to upgrade the application models by effect handlers (a component that is able to map changes in the WebSpec level to technology ones), the plugin supports the generation of classes and methods compatible with Seaside and GWT, and we are actively working to provide a derivation to WebRatio design models [37].

As an example of the use of effect handlers, we next show how to use the change objects of our exemplar upgrade (Add a register functionality) to generate classes and methods in GWT. For the sake of conciseness we show the new RegisterView class created by the GWT effect handler. Basically, lines 09-15 define the instance variables representing the widgets, and lines 21-29 initialize the objects with the proper GWT classes. Also, notice that RegisterView extends VerticalPanel (a GWT base class for implementing UIs).

```
(01) package org.webspeclanguage.re;
(02)
(03) import com.google.gwt.user.client.ui.VerticalPanel;
(04) import com.google.gwt.user.client.ui.TextBox;
(05) import com.google.gwt.user.client.ui.Button;
(06)
(07) public class RegisterView extends VerticalPanel {
(08)
(09)     private TextBox firstName;
(10)     private TextBox lastName;
(11)     private TextBox username;
(12)     private TextBox password;
(13)     private TextBox confirmPassword;
(14)     private Button register;
(15)     private Button cancel;
(16)
(17)     public RegisterView() {
(18)         this.initializeComponent();
(19)     }
(20)
(21)     public void initializeComponent() {
(22)         this.firstName = new TextBox();
(23)         this.lastName = new TextBox();
(24)         this.username = new TextBox();
(25)         this.password = new TextBox();
(26)         this.confirmPassword = new TextBox();
(27)         this.register = new Button();
(28)         this.cancel = new Button();
(29)     }
(30) }
```

5 Case study

5.1 Introduction

We have used the WebSpec plugin to assist the development of an application for the post-graduate area of the College of Medicine in the University of La Plata. The development team is composed of 2 developers, 1 analyst and a project manager using as a development approach an updated version of WebTDD [30] (suitable for code-based development). WebTDD is an agile test-driven development approach with strong emphasis on using mockups and tests to drive the development process.

The requirements were obtained from one person (the head of the college) thus avoiding any conflict resulting between different stakeholders. The project was divided in sprints (as in most agile approaches) in which we tackle a set of requirements delivering a running application to the customer. In our case we had 6 sprints to implement several user stories though here we only show the first 3 sprints. Each sprint was delivered within 2 weeks thus gathering quick feedback from the customer. The first 3 sprints tackle the following user stories:

- **Sprint 1**
 - *Login*: As a user, I would like to login in the application using my gmail account.
 - *Log out*: As a user, I would like to log out from the application.
 - *Create user*: As an administrator, I would like to create users with roles of administrators or doctors.
- **Sprint 2**
 - *Create patient*: As an administrator, I would like to create new patients describing their personal information.
 - *Create hospitalization*: As an administrator, I would like to create a hospitalization for a patient and assigning it to an existing doctor.
 - *Update patient status*: As a doctor, I would like to update the status of the patient according to its vital signs.
 - *Close hospitalization*: As an administrator, I would like to close a hospitalization when a patient leaves the hospital.

- **Sprint 3**
 - *Notify doctor about pending patient status:* As an administrator, I would like to notify a doctor by email when it forgets to update a patient status.
 - *Update patient:* As a doctor, I would like to be able to update the patients' personal information.
 - *Assign doctor to hospitalization:* As an administrator, I would like to change the assignation of a patient to a doctor.
 - *Report about patients by doctor:* As an administrator, I would like to see a report about how many patients have been attended by each doctor filtering by dates, doctor and sex.

5.2 WebSpec use

WebSpec was used across the development cycle to specify the whole set of requirements since they all involved with interaction features. For each user story, we created a set of WebSpec diagrams to specify them and in some cases such as “Notify doctor about pending patient status” we have added some notes to the diagram to specify behavior not perceived from the UI (e.g. sending emails). Mockups were used in conjunction with WebSpec only on the first sprint mainly to define the UI of the application. On the other hand, the test suite that was obtained from the diagrams and grew along the sprints was used to drive the development cycle and to avoid breaking existing functionality. As an example, in Fig. 23 we show the diagram for the “Notify doctor about pending patient status” diagram.

Since WebSpec already provides derivation to GWT, we have used a solution based on the following technologies to implement the application: GWT, Spring and Hibernate. We took advantage of the automatic evolution of the structural part of the UI classes handled by WebSpec and therefore we only needed to code those aspects related with UI behavior and business logic.

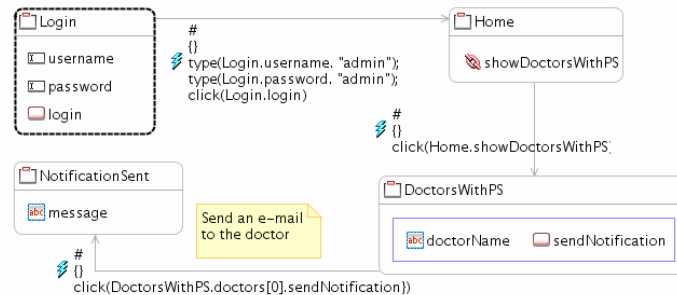


Figure 23. Notify doctor about pending patient status

As a summary, Table 1 shows for each sprint the number of user stories per sprint, the number of test cases obtained from the diagrams and if simulation and code generation was used or not in the sprint. We must notice that we didn't use code generation in the last sprint as it was a behavioral change which can not be automated by the GWT effect handler. Also, because of the nature of the application, simulation was only used in the first sprint and afterwards it was not needed.

	Nro user stories	Nro WebSpec	Tests generated	Simulation?	Code generation?
Sprint 1	3	3	10	Yes	Yes
Sprint 2	4	3	14	No	Yes
Sprint 3	4	4	16	No	Yes
Sprint 4	3	4	8	No	Yes
Sprint 5	4	5	10	No	Yes
Sprint 6	2	1	7	No	No

Table 1. Summary of WebSpec use

5.3 Advantages and disadvantages

After we finished the 6 sprints of the project we conducted a survey with the customer and the development team to asses the experience of using WebSpec in the development process.

The customer liked the use of mockups and the simulation features of WebSpec as they gave him a clear picture of the understanding of the analyst regarding the requirements. Though simulation was used in the first sprint, it helped to define the base UI and behavior necessary to

build the Web application. On the other hand, some diagrams were rather complex (specially the list of actions) and thus hard to understand by the customer. He suggested providing a simplified view of the diagram in those cases.

In the development team the most appreciated feature was the test suite derived directly from the diagrams. The test suite was used to assess if the requirements were correctly implemented during the development cycle and to check that new code did not break existing functionality. The test suite grew quickly and therefore the time consumed to run the tests also grew. As a criticism to the kind of tests that WebSpec derives, the development team agrees on the necessity of interaction tests but they prefer small unit tests to be derived (A feature that WebSpec does not have yet). As an improvement, the development team created a continuous build² to run the test suite. Finally, in the coding side, mockups and WebSpec diagrams help to implement the requirement using the code derivation features (GWT effect handler) and were appreciated by developers as it automates UI changes.

In conclusion, the experience with both customers and the development team was positive though some features can be improved such as the language readability and the generation of unit tests. We expect to improve these features in future works.

6 Related Work and Discussion

As we have previously stated, the specification of interaction and navigation requirements is a complex task due to some unique characteristics of Web applications such as the need to represent the navigation in information spaces, the need of describing technical constraints related to the information flow (e.g. session management), the rapid evolution of requirements and the participation of customers and other stakeholders in the development process (e.g. marketing experts, editorial board, etc) [38]. In the last years, a large variety of artifacts have been employed to capture Web requirements like UML use cases and sequence diagrams [39], User Interaction Diagrams [4], task models [40], and navigation models [8]. It is also worth noting a widespread use of paper-based mockups to capture requirements related to the user interface of Web applications [41] which has led to the development of advanced tools for sketching and storyboarding the user interface of Web applications such as Denim [42] and Balsamiq [23].

However, existing approaches have some drawbacks: many of them are not suitable to be used as communication tools with clients, others provide very informal ways of specifying the requirements, which cannot be then validated and some others which provide partial derivation of domain or navigational models don't deal well with evolution. In the following sub-sections we survey how the most important Web engineering methodologies support the specification of requirements and compare the different requirement artifacts used.

6.1 Requirements in Model Driven Web engineering

In [9], Escalona and Koch have investigated how different Web engineering methods support the capture of requirements. They showed that some methods employ classical notations to deal with Web requirements, and others simply ignore this phase in the development process. It is interesting to notice that requirement artifacts might play several roles during the development process: they can act as communication tools (for elicitation requirements with clients), as elements for early specifications (that should be taken into account during implementation phases) and as checklists for assessing if the final implementation complies the initial requirements. Requirement checklists can indeed be employed in regression testing [15] for assessing in a longer term, the evolution of requirements expressed for a single application.

Many Web engineering methods, such as UWE [6], WebML [7], OOWS [5], OOHDM [4] and NDT [43], include UML use case diagrams for capturing requirements. However, use case diagrams are not sufficient for capturing all the details of Web application requirements. Therefore, these Web engineering methods have often included more than one artifact for capturing requirements; for example use cases are present in OOHDM in combination with UIDS [4]. Besides, use cases and activity diagrams, WebML uses semi-structured textual descriptions to capture additional information that can hardly be expressed using the former models. Similarly, UWE [6] proposes extended use cases, scenarios and glossaries for specifying requirements and OOWS [5] combines use cases with extended activity diagrams with the concept of interaction points to describe the interaction of the user with the system.

² A continuous build is a program that compiles the application and runs the tests separately without interfering in the developer's activity.

Other approaches do not consider UML diagrams, such as WSDM [10] which employs task models using concurrent task trees and A-OOH [12] which considers the i* framework [44] in order to specify the requirements model which is goal-oriented.

Some authors have investigated how to automate the generation of the system specification from the requirements specification; for example OOWS provides automatic generation of (only) navigation models from the tasks description, by means of graph transformation rules. In A-OOH the conceptual models (e.g. domain and navigation models) are generated by means of QVT [45] transformations from the requirements specification in i* models. NDT defines a requirement metamodel and allows transforming the requirements into conceptual models (content and navigation models) by using QVT rules [45].

Table 2 summarizes the most relevant development approaches and which requirements artifacts they use. We have also added a row indicating the existence of a requirement analysis tool for the process.

Approach	NDT	WebML	UWE	OOWS	WSDM	A-OOH	WebTDD
Textual templates	X						
Use cases	X	X	X	X			X
Activity diagram		X	X				
Task diagrams				X	X		
i*						X	
User stories							X
Mockups							X
WebSpec							X
Other techniques				FRT	Concept maps		
Derivation of the application or models	X		X	X		X	X
Requirements Analysis Tool	NDT-Tool	No	Magic UWE	OOWS-Suite	No	Eclipse Plugin	Eclipse plugin

Table 2. Requirements artifacts in Web engineering approaches

6.2 Requirements Artifacts and Discussion

In Table 3 we compare the expressive power of *some* artifacts with respect to the different aspects that are needed for representing Web requirements [9]. Next, we will describe and compare some other important requirement artifacts.

As shown in table 3, each artifact includes only part of the concepts required to express requirements of Web applications. For example, whilst use cases can be used to represent functional requirements, mockups (either paper-based or supported by tools) are more likely to capture and represent requirements related to the composition of the user interface. Task models allow expressing fine-grained functional requirements including navigation, user transactions and business processes.

Concept		Artifacts used for representing requirements					
		Use cases	Task Models	WebRE	WebSpec	Mockups	I* extension
Behavior	Navigation	Dependencies between UC	Dependencies between tasks	Navigation	Navigation arrows	Arrows	Navigation requirement
	Process	Use cases	Tasks,	WebProcess	WebSpec diagram	-	Service requirement
	User interaction	Functional requirements	Interactive tasks	User transaction	Action	-	-
	Constraints	OCL	Lotus operators	OCL	Precondition & Invariants	Annotated text	OCL
	Information flow	-	Data transfer between tasks	Data transfer in user transaction	Data transfer between interactions	-	

Table 3. Expressiveness power of requirement artifacts for Web applications

All these artifacts are quite similar from what they can express; however, they have different notations and may use similar concepts. In order to provide a more uniform view on the coverage of requirements by each artifact, Escalona and Koch have proposed a metamodel based on WebRE profiles [46]. Its main advantage is the automatic generation of conceptual models (content and navigation models) which automatically satisfy the requirements. Notwithstanding, some requirements such as detailed composition of the user interface and behavior constraints cannot be fully described with this notation.

Two widely used artifacts for capturing requirements in Web engineering are textual templates and use cases. Textual templates are specified in natural language in a structured way as tables with predefined fields that the designer should fill in. Natural language is ambiguous so requirements are specified in an imprecise and informal manner. Also they are difficult to fill in,

maintain and unsuitable for expression UI aspects. Use cases are also an ambiguous technique when defining complex requirements. Usually they have to be complemented with other techniques such as textual templates or activity diagrams and if special attention is needed to represent UI concepts, it should be combined with mockups or UML UI models.

As a way to overcome some of the problems of using natural language while capturing interaction aspects, User interaction diagrams [4] (UID) were proposed. UIDs help to define the interactions that the user has with the Web application. Despite the fact that they are formally defined, the actions that produce navigations are described in a non structural fashion. As a consequence, UIDs can only be used for capturing requirements and do not help to validate them. As aforementioned, requirement artifacts are not updated if they do not help during the development process thus making the validation process harder.

In the requirements engineering field, I* [44] is widely used to model the expectations, needs and goals of the users and making design decisions from the very beginning of the development phase. Recently, we have proposed an extension [12] to express navigation and UI requirements as stereotypes of goals and tasks. However, our approach is not suitable for communication with clients as requirements are describe in an abstract way and do not described precisely UI aspects. As a consequence, those details are discussed with customers too late.

In the agile track, user stories and mockups are the typical way of capturing requirements because they improve the communication with clients, since they allow specifying a prototypic user interface. The story describes informally the requirement that the client has, and sets a starting point to talk and discuss requirements with clients. If these artifacts are not combined with a test-based development approach, checking if a requirement is still satisfied by the application after several iterations would be impossible. The main drawback of using these artifacts solely is that tests are written manually and by deducing the behavior from an informal textual representation.

MoLIC [16] though not explicitly defined for the Web field, was devised to represent the human-computer interaction as the set of conversations that users may (or must) have with the system (more precisely, with the designer's deputy) to achieve their goals. The main aim of MoLIC is to support the designers' reflection on the interactive solution being conceived. As it was proposed for human usage, MoLIC is not a formal, computer-processable model. Molic diagrams are similar to WebSpec's, however WebSpec is a formal language and Molic is not. Therefore, you cannot derive a test suite or simulate the application using mockups as in WebSpec. Also, MoLIC seems good for communication with stakeholders but due to its lack of automatic features it tends to be an overhead if it is used in agile methodologies. WebSpec meanwhile can be used in both Agile and more cascade style of approaches due to its automatic features.

According to industrial studies [1, 2] one of the main problems of the current use of requirements artifacts for Web applications is that it is impossible to validate that the requirement has been implemented correctly. Therefore, we strongly believe that obtaining a test suite from the requirement specification is important to validate new and old requirements (regression testing) in the application and most important when the application grows during its life cycle. In this aspect, WebRE is the only approach that provides test derivation from the specification, though it is tightly coupled with the NDT development approach. In WebSpec we can derive a test suite that can be used with any development approach as tests are derived in Selenium. When the test suite is run it opens a Web browser and executes actions over it as a user would do it making it a technology independent approach. Even an application written manually in PHP could be validated with the tests generated from a WebSpec diagram.

As a summary, table 4 shows a comparison between the features of each requirement artifact. We have included the features that we think are needed for actively using requirements during the development cycle (simulation, test derivation and application derivation). Many of the requirement artifacts provide some type of derivation of the application; either class or model derivation. However, most of the do not help to validate that the requirements they express were correctly implemented and also to improve the interaction between the different members of the development team (simulation). With WebSpec we expect that requirements artifacts play a key and important role mainly acting as drivers during the whole development cycle.

	Use cases	Task Models	WebRE	WebSpec	Mockups	i*
Simulation				X	X	
Technology independent test derivation				X		
Derivation of the application or models		X	X	X		X

Table 4. Comparison of the features of each requirement artifact

7 Concluding Remarks and Further Work

In this paper we have presented a detailed and complete definition of the WebSpec domain specific language. Webspec allows building requirements artifacts used to capture navigation, interaction and UI features in Web applications independently of the underlying software development process.

We have shown examples of how to specify navigation and rich behaviors and we have briefly described how we can scale up when thousands of requirements are specified with WebSpec by using its composition features. We have shown how a Web application can be simulated when using WebSpec with mockups by presenting the mockups and showing descriptions over them. An interesting property of WebSpec diagrams is that test suites that validate the specified requirements are obtained automatically from the diagram (e.g. a selenium test suite). Finally, changes in the requirements are captured in change objects and then by using a specific effect handler, a set of classes/models are created or updated. In this case we have shown the code generated by the GWT effect handler. In Sect. 5, we have shown how we have used WebSpec in the context of an agile approach like WebTDD to develop an application in several sprints while deriving part of the GWT code and using the derived test to validate the correct behavior of the application.

We are currently working on several research lines to improve WebSpec both from an internal perspective (intrinsic to the language and its features) and more external (related with other approaches and development processes). Our first effort is to complete a set of testing frameworks for WebSpec, so that we can give more flexibility to development teams. These frameworks include Watir [47] and Webdriver [36]. On the other hand, we are improving support for a set of technologies to be used to automatically manage implementation changes. Right now we support Seaside (Smalltalk based) and GWT (Java based), but we are also working on PHP, Ruby and .NET.

Also, we have obtained some preliminary results on several areas that need further research. First, from an internal perspective, we proposed a small extension to specify usability in the language [48, 49] and personalization requirements by means of special variables [50]. In the first case, usability is a distinctive feature in the current competitive market to attract more users (e.g. in social networks like facebook, sonico or myspace). Allowing to express usability aspects in the diagrams help to define a test suite that the application must satisfy. On the other hand, personalization is pretty important for e-commerce applications and therefore specifying this kind of requirements is critical. Our approach is simple and lets specifying the most basic personalization scenarios. However, we are in the preliminaries of this work which needs further research for example to automate the definition of reusable personalization patterns.

Second, according to the definition of [52], WebSpec can be considered a requirement artifact that should be used on a late requirement analysis phase. Therefore, we have proposed in [51] the use of WebSpec with an early phase with i^* . Our work proposes an automatic validation algorithm of the i^* model based on the association between WebSpec and the goals and tasks of the i^* model. However, the derivation process can still be improved by mixing the derivation process of domain and navigation classes proposed in [51], with the navigation and UI derivation process of WebSpec. As a consequence, we could automatically derive the three design models that most model-driven development approaches for Web applications support (domain, navigation and UI models).

Finally, in [53] we have presented an approach to derive a complete structural UI model/class from a mockup. Though the approach is independent from WebSpec, our first experiments have shown that when used together with WebSpec, we can obtain a more complete derivation of the application.

References

1. McDonald A. and Welland R., Web Engineering in Practice, Proceedings of the Fourth WWW10 Workshop on Web Engineering, Page(s): 21-30, 1 May 2001.
2. Lowe D, Web system requirements: an overview. Journal of Requirements Engineering pp 102–113. Springer-Verlag (2003).
3. Jacobson, I., Object-Oriented Software Engineering: A Use Case Driven Approach, ACM Press/Addison-Wesley, 1992.
4. Rossi, G., Schwabe, D.: Modeling and Implementing Web Applications using OOHDM. In: Web Engineering, Modelling and Implementing Web Applications, pp. 109–155. Springer, Heidelberg (2008).

5. Valderas, P., Pelechano, V., Pastor, O. 2008. A transformational approach to produce Web applications prototypes from a Web requirements model. *Int. J. Web Engineering Technology*, IJWET 3(1): 4-42 (2007)..
6. Koch, N., Zhang, G. AND Escalona, M.J. 2006. Model Transformations from Requirements to web System Design. ICWE'06, Palo Alto, California, USA.
7. Ceri, S., Fraternali, P., Bongio, A., Brambilla M., Comai S. and Materna M. 2003. *Designing Data-Intensive web Applications*. Morgan Kaufman.
8. Gómez, J., Cachero, C.: OO-H Method: extending UML to model web interfaces. In: van Bommel, P. (ed.) *Information Modeling For internet Applications*, pp. 144–173. IGI Publishing, Hershey (2003).
9. Escalona, M.J., Koch, N.: Requirements engineering for web applications – a comparative study. *J. Web Eng.* 2(3) (2004) 193–212.
10. De Troyer, O., Casteleyn, S. Modeling Complex Processes for Web Applications using WSDM. In: 3rd Int. Workshop on Web-Oriented Software Technologies. Oviedo, Spain (2003). At: <http://www.dsic.upv.es/~west/iwwost03/articles.htm>
11. Escalona, M.J., Koch, N. Metamodeling Requirements of Web Systems. In *Proc. International Conference on Web Information System and Technologies (WEBIST 2006)*, INSTICC, 310--317, Setúbal, Portugal. 2006.
12. Garrigós, I., Mazón, J.N., Trujillo, J.: A Requirement Analysis Approach for Using i* in Web Engineering. In: ICWE. (2009), LNCS, 5648, 151-165.
13. Martin, RC. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA. 2003.
14. Beck, K.: *Test Driven Development: By Example*. Addison-Wesley Signature Series (2002).
15. Zheng, J. In regression testing selection when source code is not available. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering* (Long Beach, CA, USA, November 07 - 11, 2005). ASE '05. ACM, New York, NY, 752-755. DOI= <http://doi.acm.org/10.1145/1101908.1101997>.
16. de Paula, M. G., da Silva, B. S., Barbosa, S. D. 2005. Using an interaction model as a resource for communication in design. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, USA, April 02-07, 2005), pp 1713-1716.
17. Rossi G, Pastor O, Schwabe D, Olsina L. *Web Engineering: Modelling and Implementing Web Applications*. Human-Computer Interaction Series. Springer, London, 2008.
18. GWT. Available at: <http://code.google.com/webtoolkit/>
19. Seaside. Available at: <http://www.seaside.st/>
20. Fowler M. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional. 2010
21. Claessen K., Hughes J., “QuickCheck: a lightweight tool for random testing of Haskell programs”, *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, vol. 35, pp. 268-279, September 2000.
22. Bondy, J. A. *Graph Theory with Applications*. Elsevier Science Ltd. 1976.
23. Balsamiq. Available at: <http://www.balsamiq.com/products/mockups>
24. Axure - Wireframes, Prototypes, Specifications. Available at: <http://www.axure.com/>.
25. Chomsky N. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, January 2003.
26. Duhl, J. *Rich Internet Applications*. A white paper sponsored by Macromedia and Intel, IDC Report, 2003.
27. Yahoo Patterns, <http://developer.yahoo.com/ypatterns/>.
28. Moody D, "The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Transactions on Software Engineering*, pp. 756-779, November/December, 2009
29. Maximilien, E. M. and Williams, L. 2003. Assessing test-driven development at IBM. In *Proceedings of the 25th international Conference on Software Engineering* (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 564-569
30. Robles Luna, E., Grigera, J., and Rossi, G. 2009. Bridging Test and Model-Driven Approaches in Web Engineering. In *Proceedings of the 9th international Conference on Web Engineering*. Lecture Notes In Computer Science, vol. 5648. Springer-Verlag, Berlin, Heidelberg, 136-150.
31. Eclipse EMF. Available at: <http://www.eclipse.org/modeling/emf/>.
32. Eclipse GMF. Available at: <http://www.eclipse.org/modeling/gmp/>.
33. Selenium web application testing system. Available at: <http://seleniumhq.org/>.

34. jQuery: The Write Less, Do More, JavaScript Library. Available at: <http://jquery.com/>
35. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1995.
36. WebDriver. Available at: <http://webdriver.googlecode.com>
37. The WebRatio Tool Suite. Available at: <http://www.webratio.com>.
38. Uden, L., Valderas, P., Pastor, O. An Activity-theory-based to analyse Web applications requirements. Information Research Vol. 13, N. 2. June 2008.
39. Conallen, J., Building Web Applications with UML, Addison-Wesley, 2000, 300 p.
40. Winckler, M.; Vanderdonck, J. Towards a User-Centered Design of Web Applications based on a Task Model. In Proceedings of IWOST'2005. Porto, Portugal, June 12-13th 2005.
41. Flannagan, S. The Paper Version of the Web. In Deeplinking, available at: <http://deeplinking.net/paper-web/>
42. Lin, J., Newman, M. W., Hong, J. I., and Landay, J. A. 2000. DENIM: finding a tighter fit between tools and practice for Web site design. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (The Hague, The Netherlands, April 01 - 06, 2000). CHI '2000. ACM, New York, NY, 510-517.
43. Escalona, M.J., Aragon, G. 2008. NDT. A Model-Driven approach for web requirements. IEEE Transaction on Software Engineering, 34(3), pp. 370-390.
44. Eric S. K. Yu. 1997. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE '97). IEEE Computer Society, Washington, DC, USA, 226.
45. QVT. <http://www.omg.org/spec/QVT/>
46. Escalona, M.J., Koch, N. Metamodeling Requirements of Web Systems. In Proc. Internacional Conference on Web Information System and Technologies (WEBIST 2006), INSTICC, 310--317, Setúbal, Portugal. 2006.
47. Watir. Available at: <http://watir.com/>.
48. Robles Luna E., Panach J.I., Grigera J., Rossi G., Pastor O. Incorporating Usability Requirements in a Test/Model-Driven Web Engineering Approach. Journal of Web Engineering (JWE). 2010.
49. Robles Luna E., Rossi G., Burella J., Grigera J. Incremental Usability Improvement in an Agile Approach for Web Applications. Proceedings of the 1st workshop Dealing with Usability in an Agile Domain, XP'2010 workshop, 2010. Trondheim, Norway.
50. Robles Luna E., Garrigos I., Rossi G. Capturing and Validating Personalization Requirements in Web Applications. Proceedings of the 1st Workshop on The Web and Requirements Engineering (WeRE 2010). Sydney, Australia.
51. Robles Luna E., Garrigos I., Mazon J-N., Trujillo J., Rossi G. An i*-based Approach for Modeling and Testing Web Requirements. Journal of Web Engineering (JWE). 2010.
52. Alencar, F. M. R.; Castro, J. F.B.; "Integrating Early and Late-Phase Requirements: a Factory Case Study". Proceedings of XIII Brazilian Symposium on Software Engineering - SBES99. Florianópolis, SC, Brasil, Outubro 1999. pp 47-61.
53. Rivero J.M., Rossi G., Grigera J., Burella J., Robles Luna E., Gordillo S. From mockups to user interface models: An extensible model driven approach. Proceedings of the 6th Model-Driven Web Engineering Workshop. (MDWE 2010). Vienna, Austria.

A WebSpec's grammar

Helpers

```

letter = [['a' .. 'z'] + ['A' .. 'Z']];
digit = ['0' .. '9'];
whitespace = ' ';
varh = '$';
left_braceh = '{';
right_braceh = '}';

```

Tokens

```

string_type = 'String';
number_type = 'Number';
boolean_type = 'Boolean';

add = '+';
sub = '-';
mul = '*';
div = '/';
var = varh;
left_brace = left_braceh;
right_brace = right_braceh;
greater = '>';
greater_equal = '>=';
not_equal = '!=';
equal = '=';
lower = '<';
lower_equal = '<=';

and = '&&';
implies = '->';
or = '||';
not = '!';

concat = '#';
left_paren = '(';
right_paren = ')';
number = (digit)+ ('.' (digit)+)?;
array_index = (digit)+;
true = 'true';
false = 'false';
whitespaces = (whitespace)+;
identifier = (letter | '_' | digit)*;
string = (" | '') ('@' | ':' | '/' | '.' | letter | digit | whitespace | left_braceh | varh | right_braceh)* (" |
"");
point = '.';
semicolon = ';';
comma = ',';
assign = ':=';
left_block = '[';
right_block = ']';
percent = '%';

```

Ignored Tokens

```
whitespaces;
```

Productions

```

actions =
    {singleaction} action
    | {manyactions} action semicolon actions;

```

```

action =
    {let} type? identifier assign [expr]:expr
    | {expr} expr ;

```

```

arguments =
    {onearg} expr
    | {manyargs} expr comma arguments;

```

```
expr =
```

```

    {and} [left]:expr and [right]:comp_expr
  | {or} [left]:expr or [right]:comp_expr
  | {implies} [left]:expr implies [right]:comp_expr
  | {not} not [comp_expr]:comp_expr
  | {comp_expr} comp_expr;

comp_expr =
  {greater} [left]:comp_expr greater [right]:num_expr
  | {greater_equal} [left]:comp_expr greater_equal [right]:num_expr
  | {not_equal} [left]:comp_expr not_equal [right]:num_expr
  | {equal} [left]:comp_expr equal [right]:num_expr
  | {lower} [left]:comp_expr lower [right]:num_expr
  | {lower_equal} [left]:comp_expr lower_equal [right]:num_expr
  | {num_expr} num_expr;

num_expr =
  {add} [left]:num_expr add [right]:factor
  | {sub} [left]:num_expr sub [right]:factor
  | {factor} factor;

factor =
  {mul} [left]:factor mul [right]:value
  | {div} [left]:factor div [right]:value
  | {concat} [left]:factor concat [right]:value
  | {value} value;

value =
  {number} number
  | {string} string
  | {boolean} boolean
  | {functioncall} identifier left_paren arguments? right_paren
  | {variable} variable
  | {generator} [left]:var identifier [right]:var
  | {parens} left_paren expr right_paren
  | {nativefunctioncall} percent identifier left_paren arguments? right_paren
  | {array} array
  | {array_access} variableorliteralarray left_block expr right_block
  | {widget_path} [interaction]:identifier
[widgets]:widget_or_widget_access_list_with_property+;

variableorliteralarray =
  {variable} variable
  | {array} array;

array = left_block arguments right_block;

variable = [left]:var left_brace [i]:identifier right_brace;

widget_or_widget_access_list_with_property = [p]:point widget_or_widget_access;

widget_or_widget_access =
  {simplewidget} [widget]:identifier
  | {widgetarrayaccess} [widget]:identifier left_block expr right_block;

boolean =
  {true} true
  | {false} false;

type =
  {string_type} string_type
  | {number_type} number_type
  | {boolean_type} boolean_type;

```


Reunido el Tribunal que suscribe en el día de la fecha acordó otorgar, por _____ a la Tesis
Doctoral de Don/Dña. Esteban Robles Luna la calificación de _____.

Alicante _____ de _____ de _____

El Secretario,

El Presidente,

UNIVERSIDAD DE ALICANTE
CEDIP

La presente Tesis de D. Esteban Robles Luna ha sido registrada con el nº _____ del
registro de entrada correspondiente.

Alicante ____ de _____ de 2011

El Encargado del Registro,

