# Capture and Evolution of Web Requirements Using WebSpec

Esteban Robles Luna[1,2], Irene Garrigós[3]
Julián Grigera[1], and Marco Winckler[4]

[1] LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
{esteban.robles,julian.grigera}@lifia.info.unlp.edu.ar
[2] Also at Conicet
[3] Lucentia Research Group, DLSI, University of Alicante, Spain
igarrigos@dlsi.ua.es
[4] IRIT, 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.

B. Benatallah et al. (Eds.): ICWE 2010, LNCS 6189, pp. 173–188, 2010.
© Springer-Verlag Berlin Heidelberg 2010

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.

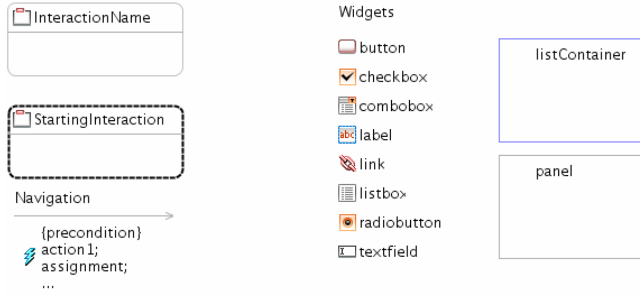


**Fig. 1.** WebSpec's basic concepts

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.

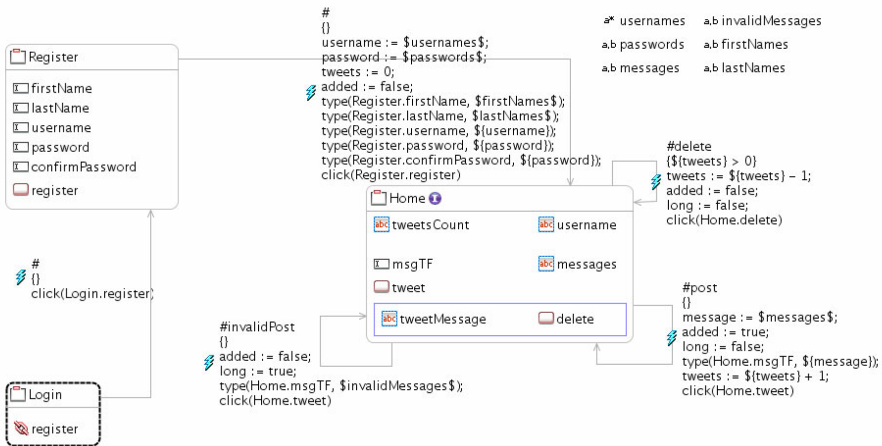


**Fig. 2.** Tweet Webspec diagram

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} && ${long} -> Home.messages = "Invalid message"* that states that the contents of the username label must be equal to the username variable (denoted as *${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 | actionName(arg1,... 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.

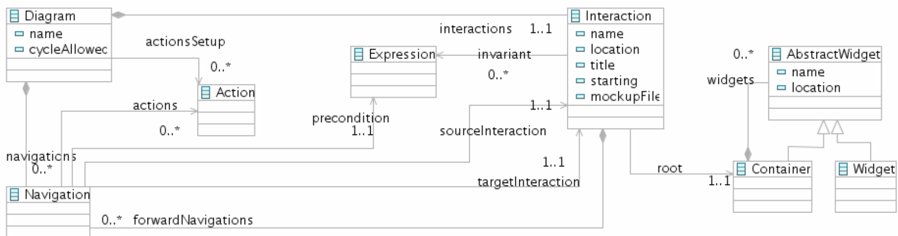


**Fig. 3.** WebSpec simplified metamodel

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.

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.

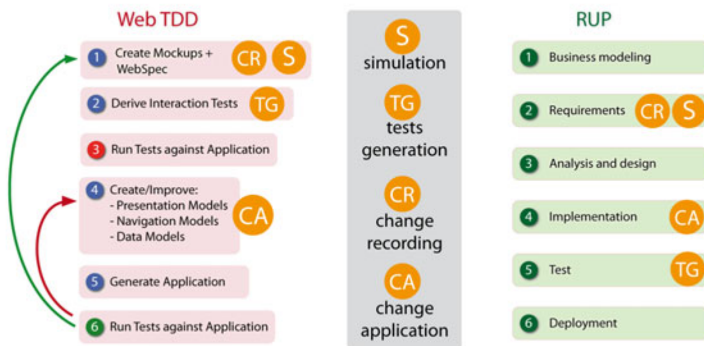


**Fig. 4.** Using WebSpec in activities of different approaches

### 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).

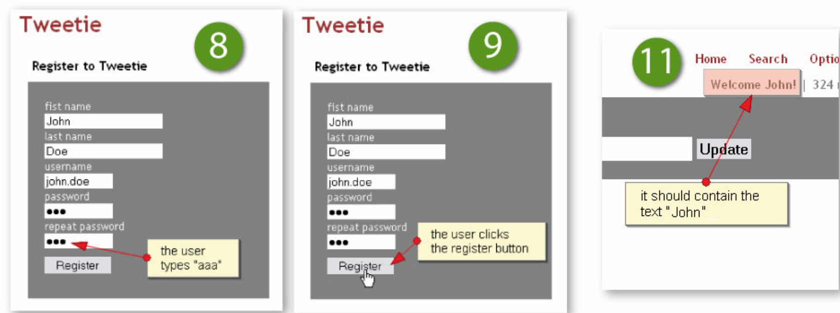


**Fig. 5.** Simulation steps of the Tweet diagram

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.

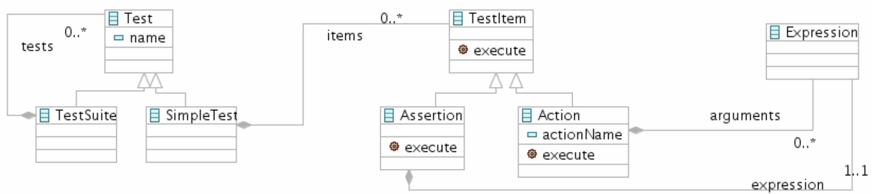


**Fig. 6.** Test metamodel

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).

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

**Fig. 7.** Change metamodel

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.

**Fig. 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

```
WALayoutPane subclass: #WATermsOfService
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Twitter'
```

New link at Login

```
createTermsOfService
| control |
control := SFAnchor link: [ self termsLinkPressed ].
control name: 'terms';label: 'terms'.
^ control
```

New label at TermsOfService

```
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
```

**Fig. 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.

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

**Fig. 10.** Webspec Eclipse plugin

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].

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

**Table 1.** Expressiveness power of requirement artefacts for Web applications

| Concept | | Artefacts used for representing requirements | | | | |
|---|---|---|---|---|---|---|
| | | **Use cases (UC)** | **Task Models** | **WebRE** | **WebSpec** | **Mockups** |
| 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 | - |
| Structure | **Node / page** | - | - | Node | Interactions / navigations | Prototype |
| | **Content** | - | - | Content | Widgets | Widgets |
| | **UI composition** | - | - | - | Containers | Prototype |
| | **User roles** | Actor | Actor | WebUser | - | - |

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, 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 (2000)
3. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: 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, 300 p . Addison-Wesley, Reading (2000)
5. de Paula, M.G., da Silva, B.S., Barbosa, S.D.: 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, pp. 1713–1716 (2005)
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), `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), 193–212 (2004)
8. Escalona, M.J., Koch, N.: Metamodeling Requirements of Web Systems. In: Proc. International Conference on Web Information System and Technologies (WEBIST 2006), INSTICC, Setúbal, Portugal, pp. 310–317 (2006)
9. Flannagan, S.: The Paper Version of the Web. In: Deeplinking, `http://deeplinking.net/paper-web/`
10. Garrigós, I., Mazón, J.N., Trujillo, J.: A Requirement Analysis Approach for Using i* in Web Engineering. In: Gaedke, M., Grissnikalus, M., Diaz, O. (eds.) ICWE 2004. LNCS, vol. 5648, pp. 151–165. Springer, Hidleberg (2009)
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, `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.: The Rational Unified Process: an Introduction, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2003)
16. McDonald, A., Welland, R.: Web Engineering in Practice. In: Proceedings of the Fourth WWW10 Workshop on Web Engineering, pp. 21–30 (May 1, 2001)
17. 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, Washington (2003)
18. Lin, J., Newman, M.W., Hong, J.I., Landay, J.A.: 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, CHI 2000, The Hague, The Netherlands, April 01 - 06, pp. 510–517. ACM, New York (2000)
19. Lowe, D.: Web system requirements: an overview. Journal of Requirements Engineering, 102–113 (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., Rossi, G.: Bridging Test and Model-Driven Approaches in Web Engineering. In: Gaedke, M., Grissnikalus, M., Diaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 136–150. Springer, Heidelberg (2009)
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, http://www.seaside.st/
24. Selenium web application testing system, http://seleniumhq.org/
25. The WebRatio Tool Suite, http://www.webratio.com
26. Uden, L., Valderas, P., Pastor, O.: An Activity-theory-based to analyse Web applications requirements. Information Research 13(2) (June 2008)
27. Winckler, M., Vanderdonct, J.: Towards a User-Centered Design of Web Applications based on a Task Model. In: Proceedings of IWWOST 2005, Porto, Portugal, June 12-13 (2005)
28. 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, ASE '05, Long Beach, CA, USA, November 07-11, pp. 752–755. ACM, New York (2005), doi:http://doi.acm.org/10.1145/1101908.1101997