Requirements Eng (2011) 16:297–321
DOI 10.1007/s00766-011-0124-1

ORIGINAL ARTICLE

WebSpec: a visual language for specifying interaction and navigation requirements in web applications

Esteban Robles Luna · Gustavo Rossi · Irene Garrigós

Received: 11 January 2011 / Accepted: 29 May 2011 / Published online: 24 June 2011

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.

E. Robles Luna (✉)
LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
e-mail: esteban.robles@lifia.info.unlp.edu.ar

E. Robles Luna
CICPBA, Buenos Aires, Argentina

G. Rossi
Conicet, La Plata, Argentina
e-mail: gustavo@lifia.info.unlp.edu.ar

I. Garrigós
Lucentia Research Group, DLSI, University of Alicante, Alicante, Spain
e-mail: igarrigos@dlsi.ua.es

Keywords Web requirements · Interaction · Testing · Simulation · Code generation

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) makes 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 [4–8], 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 fulfills the requirements and it might be difficult to handle fast evolution.

Existing languages to model Web requirements, e.g., user interaction diagrams [4] and extended use cases [10], 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 multipurpose 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 Sect. 2, we present WebSpec, its concepts, and syntax. In Sect. 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 Sect. 5, we present a case of study showing how WebSpec has been used for the development of a Web application for the postgraduate area of the College of Medicine in the University of La Plata. Section 6 presents related work, and finally in Sect. 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 that 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 [4, 16] 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 that 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 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.

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.

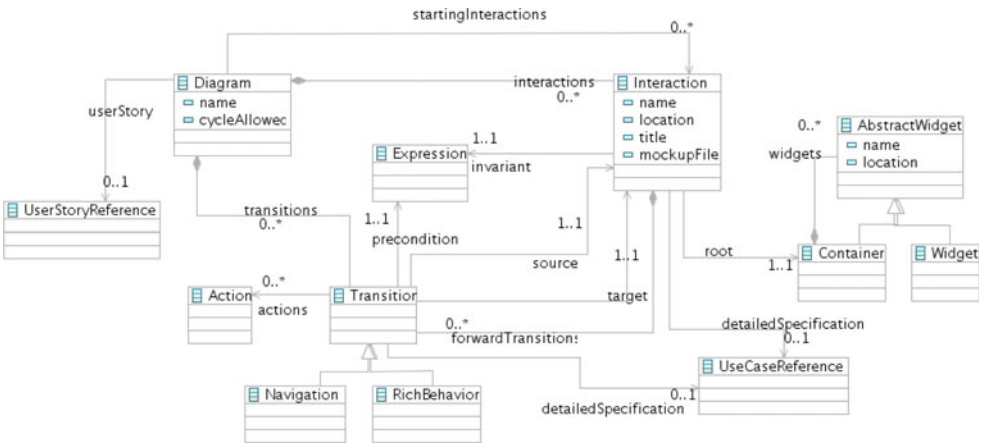


Fig. 1 WebSpec simplified metamodel

Figure 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 that 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.

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 subsections, 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.

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 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

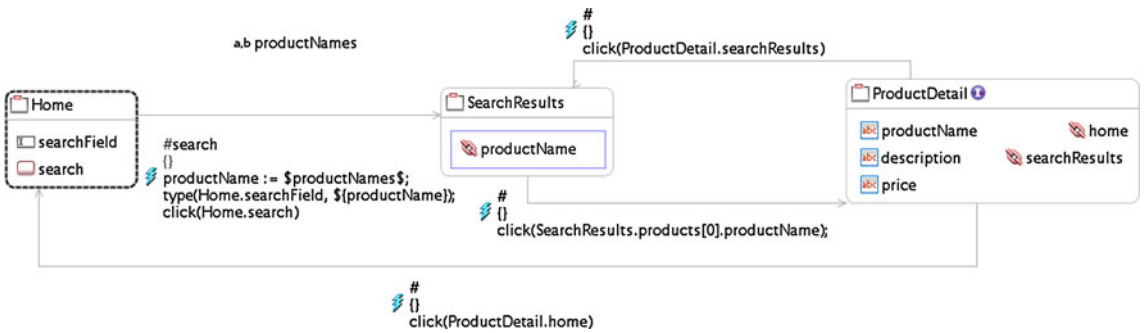
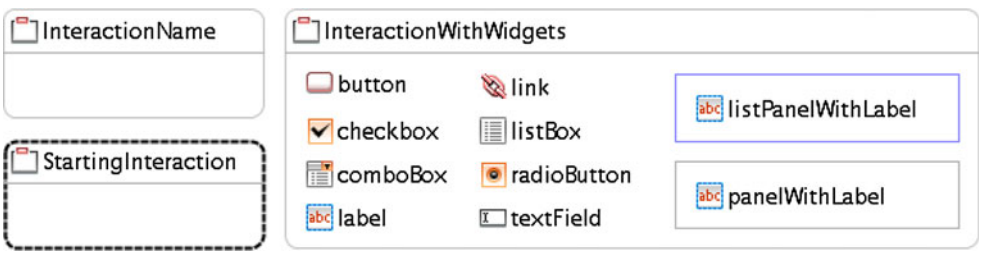


Fig. 2 Webspec diagram of the *Search by name* scenario

Fig. 3 WebSpec's interaction



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 (&&), or (||), implications (→) and negations (!). Please, refer to the "Appendix" 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).

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. Figure 5 shows a simple association between the mockup of Fig. 4 with its corresponding *interaction* and widgets of Fig. 2.

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

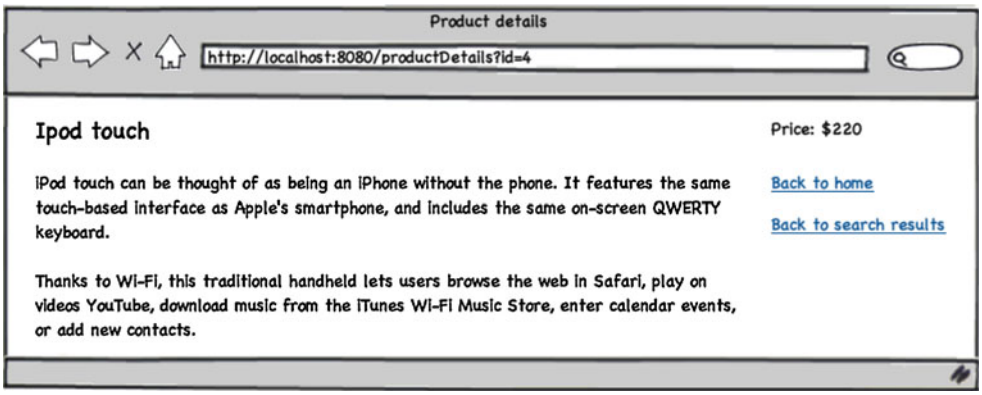


Fig. 4 Product details mockup created with Balsamiq

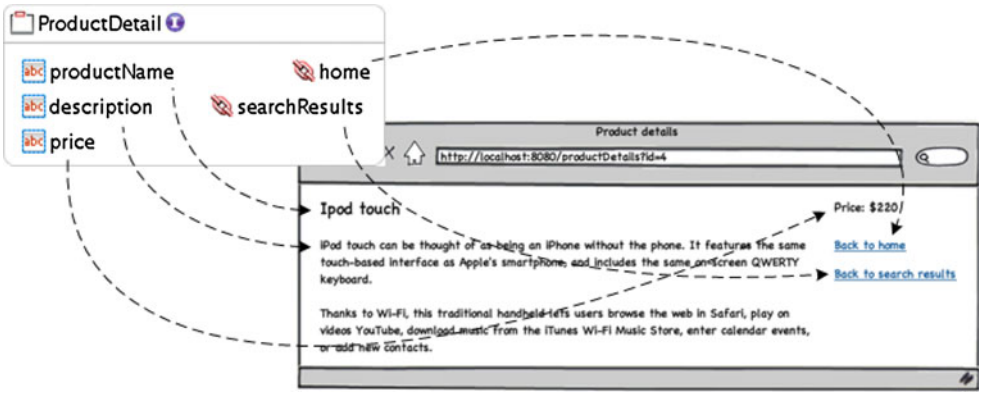


Fig. 5 Association between a mockup and its corresponding interaction

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".

In WebSpec, a *navigation* is graphically represented (Fig. 6) with gray 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 and adding some text in a text field. 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").

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 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 it. 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*).

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

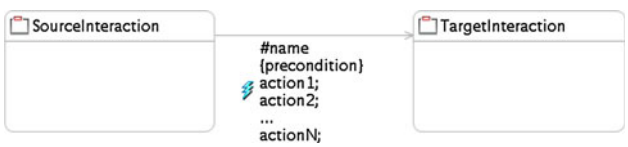


Fig. 6 WebSpec's navigation

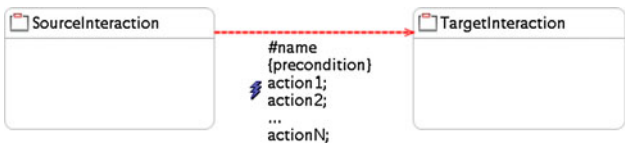


Fig. 7 Rich behavior specification in WebSpec

Fig. 8 Hover detail in *SearchResults* interaction

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. Figure 9 adds a note to the search *navigation* to explain that the search operation should be implemented by integrating with Google Search.

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 and format of exchange data. 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"*). One 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 least 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.

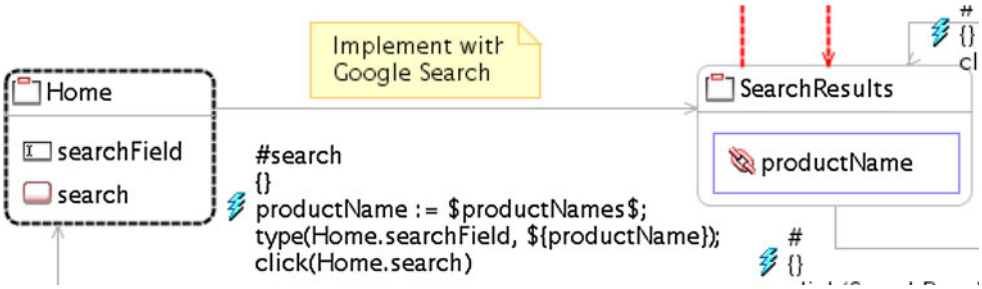


Fig. 9 Note explaining *Search* implementation details

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., $productNames$) 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.

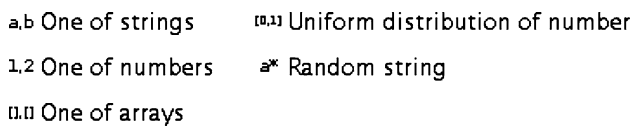


Fig. 10 Different types of WebSpec's generators

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. Figure 11a and b shows the WebSpec diagrams corresponding to each requirement.

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 subpath—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 subpath 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. Figure 12 shows the definition of the LoginAsAdminAndSearchForUser operation.

As a consequence, the diagrams of Fig. 11a and b can be written in a more short way as shown in Fig. 13a and b. These diagrams are the composition between the LoginAsAdminAndSearchForUser operation and the subpaths of Fig. 11a and b.

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.

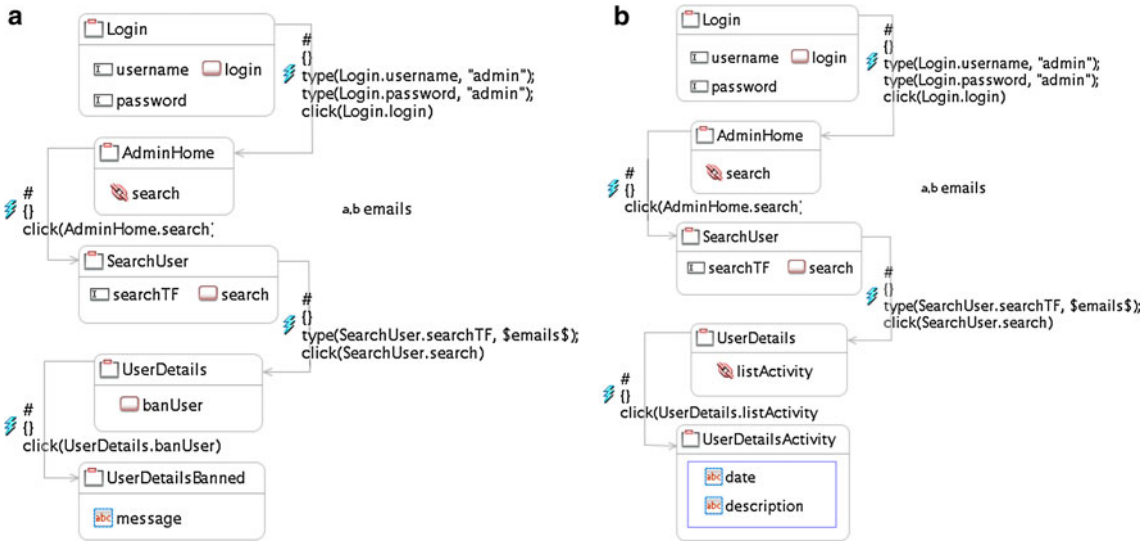


Fig. 11 **a** *Ban user* diagram. **b** *Check User's activity* diagram

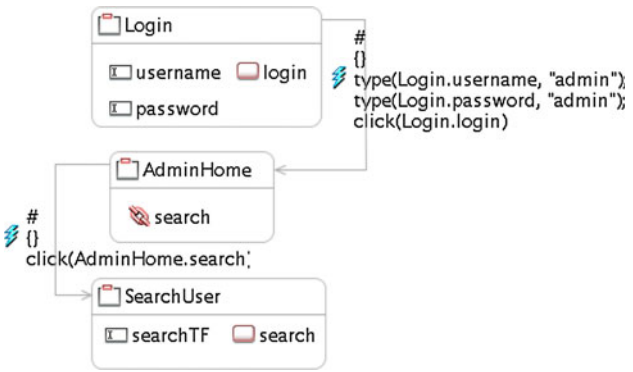


Fig. 12 *LoginAsAdminAndSearchForUser* operation

- *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).

As an example, we have added a new diagram to the one in Fig. 2 that specifies the register User story. Figure 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).

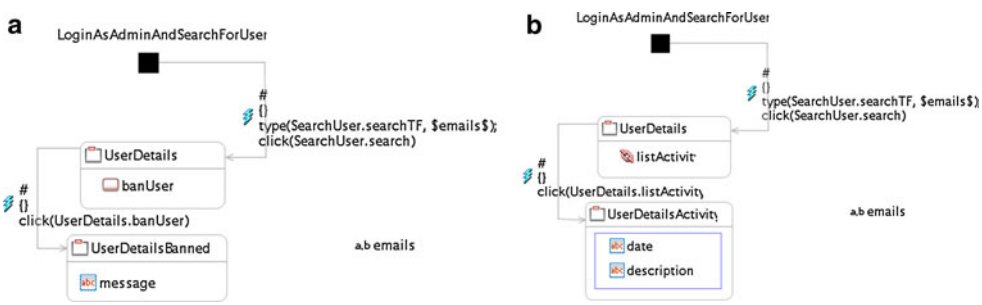


Fig. 13 **a** Refactored *Ban user* diagram. **b** Refactored *Check User's activity* diagram

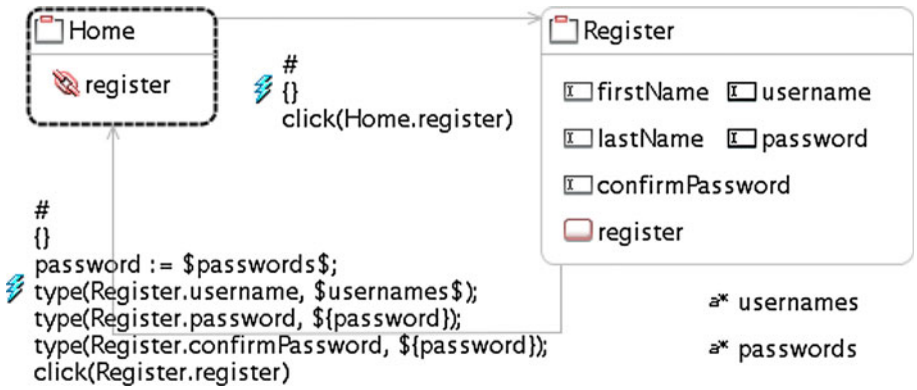


Fig. 14 *Register* WebSpec's diagram

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 two 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 shows 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 that can be used at any time by customers to understand the meaning of the diagram and suggest changes or improvements to the analyst.

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. Afterward, 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).

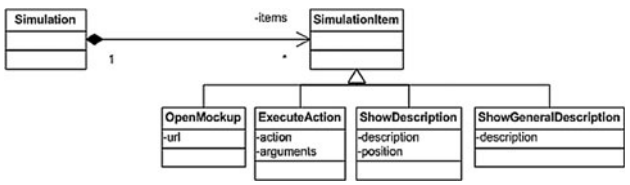


Fig. 15 Simulation metamodel

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.

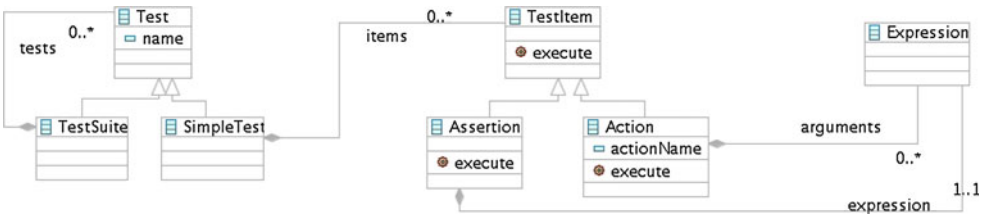


Fig. 16 Test metamodel

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).

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*.

In the following section, we show how changes in the requirements help to upgrade the application under development to satisfy the new requirements.

Fig. 17 Change metamodel

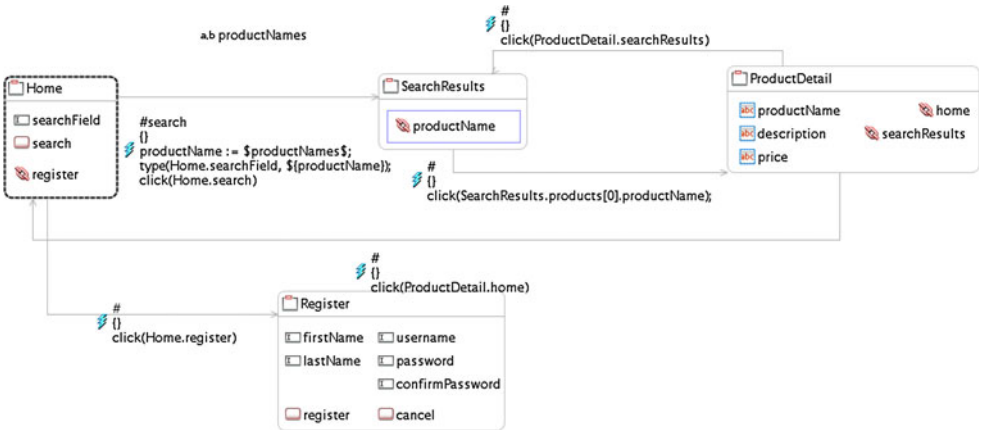


Fig. 18 Adding a register page to our E-commerce application

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.

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.

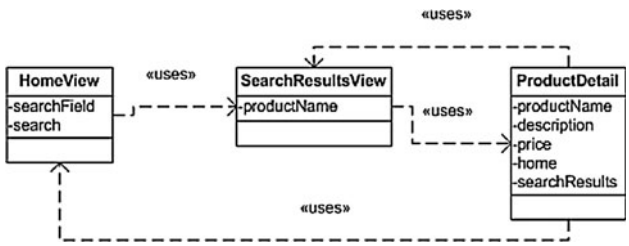


Fig. 19 Class diagram of the UI model before applying the change

- 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.

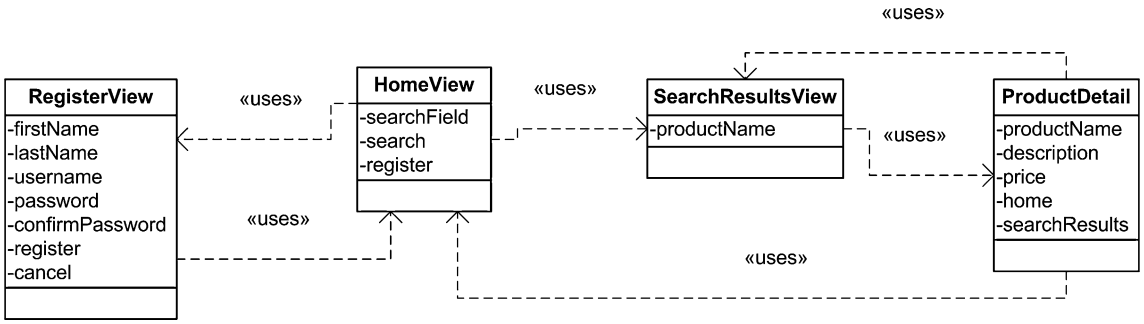


Fig. 20 Upgraded version of the UI model after applying the change

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 project.[1] 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 shows 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.

Figure 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.

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.

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

[1] http://code.google.com/p/webspec-language/.

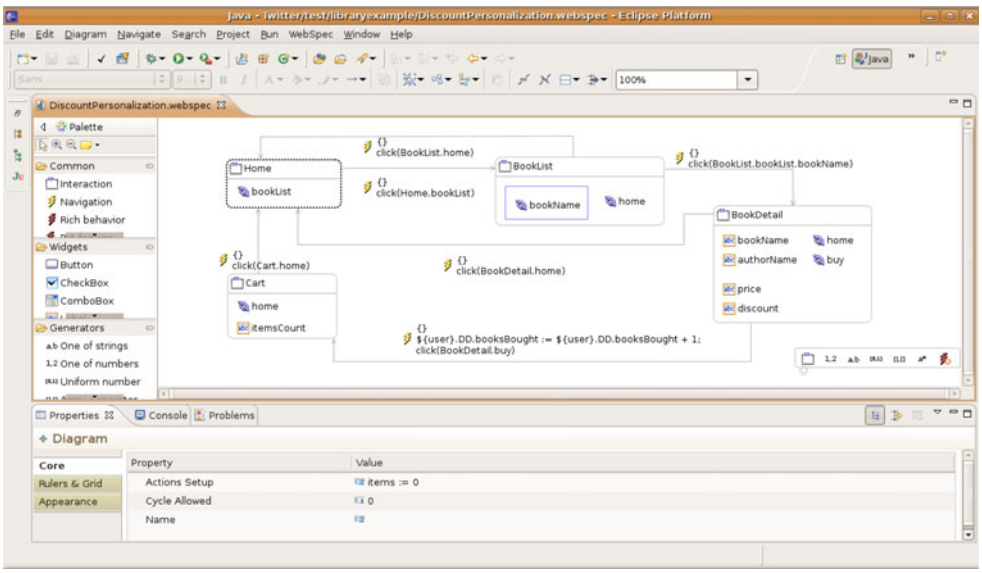


Fig. 21 WebSpec's eclipse plugin

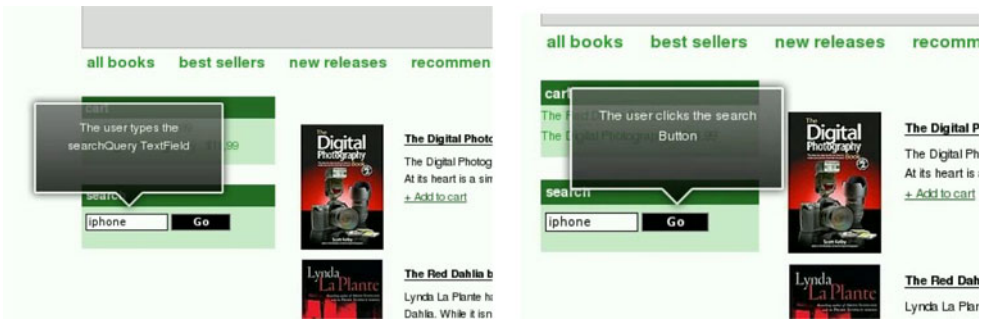


Fig. 22 WebSpec's simulation

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 select 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 Sect. 3.4.1 in XML format. To capture the changes, we use the observer pattern and incrementally build the change model; afterward, 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 postgraduate area of the College of Medicine in the University of La Plata. The development team is composed of two developers, one 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 six sprints to implement several user stories though here we only show the first three sprints. Each sprint was delivered within 2 weeks, thus gathering quick feedback from the customer. The first three 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. 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.

As an example of the use of WebSpec, we show how a requirement is captured, implemented, and validated using the formalism in our case study; we have chosen one requirement in which we have used all of the language's features: "As an administrator, I would like to create users with roles of administrators or doctors". We start the development cycle creating mockups and specifying the user story using WebSpec in an iterative basis; we activated WebSpec's change management to take advantage and derive the GWT classes using our GWT effect handler. After a few iterations, we obtained the diagram of Fig. 23 and the mockups of Fig. 24.

As the application was not built yet, we used simulation to double check the expected behaviors with our customer before implementing them; we show the first four steps in Fig. 25.

Once the requirement has been captured in the diagram and its changes have been captured in a change object, we are ready to implement it. First, we apply the change using the GWT effect handler; we obtain/update the classes of our code. As an example, we next show in Fig. 26 the code derived for the Login interaction using our GWT effect handler.

The effect handler generates the UI part, and then, we need to complete the implementation manually, till the application satisfies the requirement. Finally, we need to check that the requirement is implemented correctly; therefore, we obtain a test suite from the diagram and run it against the application. In Fig. 27, we can see the test suite obtained from the diagram of Fig. 23.

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 were used or not in the sprint.

Simulation was only used in the first sprint to improve the understanding of the diagrams and show how the application is going to behave. It was necessary at the beginning as the customer was not able to understand the diagrams and helped to reduce the semantic gap between WebSpec and the application under development (from the customer's point of view). After the first sprint,

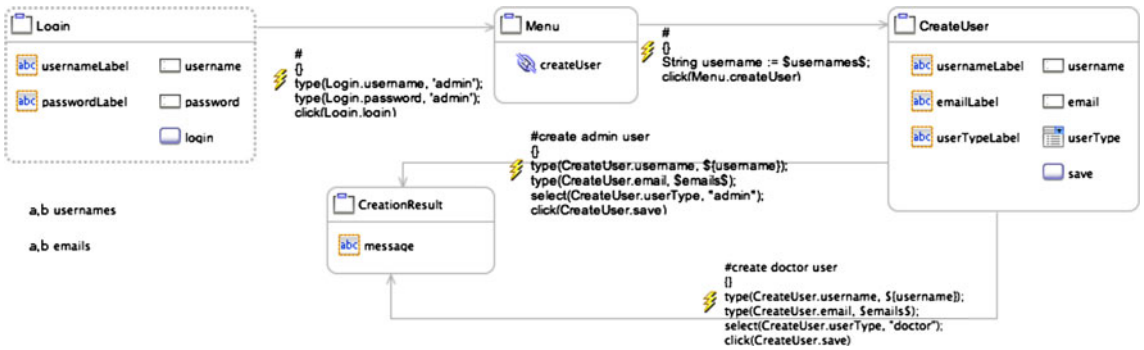


Fig. 23 WebSpec diagram of our selected user story

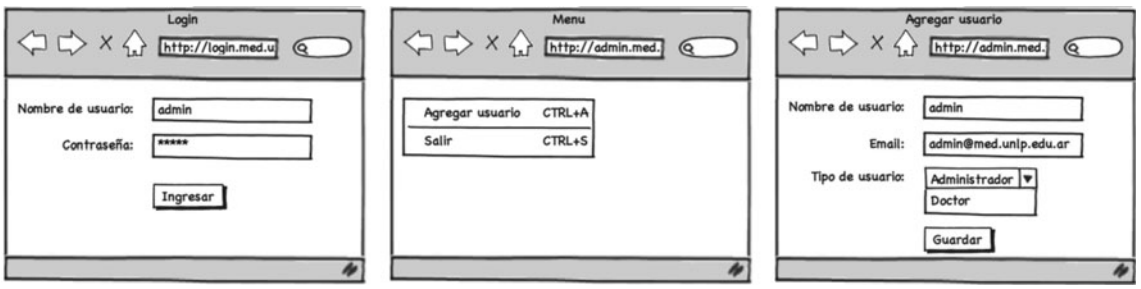


Fig. 24 Mockups of our selected user story

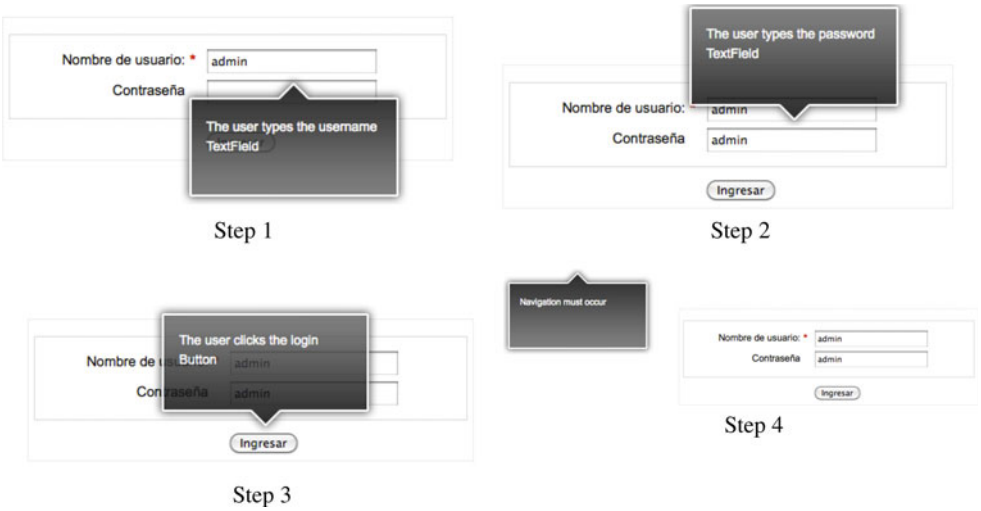


Fig. 25 First 4 steps of the simulation of our selected user story

```
package ar.edu.unlp.med.doctor;

import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

@SuppressWarnings("unused")
public class LoginView extends VerticalPanel {

    private Label usernameLabel;
    private Label passwordLabel;
    private TextBox username;
    private TextBox password;
    private Button login;

    public LoginView() {
        this.initializeComponent();
    }

    public void initializeComponent() {
        this.usernameLabel = new Label();
        this.passwordLabel = new Label();
        this.username = new TextBox();
        this.password = new TextBox();
        this.login = new Button();
    }
}
```

Fig. 26 LoginView GWT class generated by effect handler

the customer had the basic terminology and was able to follow a diagram without the necessity of simulation. This occurred in part because of the nature of the application we were building: an application with small exposure to external users and with relative simple navigational behavior.

We must notice that we did not use code generation in the last sprint as it was a behavioral change that cannot be automated by the GWT effect handler. The changed was performed manually in the core business logic of the application and checked with the test obtained from the diagrams.

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 asses whether 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 agree 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 build[2] 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.

[2] A continuous build is a program that compiles the application and runs the tests separately without interfering in the developer's activity.

Fig. 27 Test suite generated from the CreateUser diagram

```
package ar.edu.unlp.med.doctor;

import com.thoughtworks.selenium.SeleneseTestCase;

public class CreateUserTestCase extends SeleneseTestCase {

    public void setUp() throws Exception {
        super.setUp();
        selenium.open("http://med.unlp.edu.ar/doctors/index.htm");
    }

    public void testLogin_Menu_CreateUser_0CreationResult() throws Exception {
        selenium.type("id=username", "admin");
        selenium.type("id=password", "admin");
        selenium.click("id=login");
        selenium.waitForPageToLoad("30000");
        selenium.click("id=createUser");
        selenium.waitForPageToLoad("30000");
        selenium.type("id=username", "d");
        selenium.type("id=email", "b@b.com");
        selenium.select("id=userType", "admin");
        selenium.click("id=save");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.getText("id=message").equals("The username d has been created"));
    }

    public void testLogin_Menu_CreateUser_1CreationResult() throws Exception {
        selenium.type("id=username", "admin");
        selenium.type("id=password", "admin");
        selenium.click("id=login");
        selenium.waitForPageToLoad("30000");
        selenium.click("id=createUser");
        selenium.waitForPageToLoad("30000");
        selenium.type("id=username", "peter");
        selenium.type("id=email", "a@a.com");
        selenium.select("id=userType", "doctor");
        selenium.click("id=save");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.getText("id=message").equals("The username peter has been created"));
    }
}
```

Table 1 Summary of WebSpec use

	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

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 lead 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 that provide partial derivation of domain or navigational models do not deal well with evolution. In the following subsections, 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 whether 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.

Other approaches do not consider UML diagrams, such as WSDM [10] that employs task models using concurrent task trees and A-OOH [12] that 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.

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.

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.

Table 2 Requirements artifacts in Web engineering approaches

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	MagicUWE	OOWS-suite	No	Eclipse plugin	Eclipse plugin

Table 3 Expressiveness power of requirement artifacts for Web applications

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 and invariants	Annotated text	OCL
Information flow	–	Data transfer between tasks	Data transfer in user transaction	Data transfer between interactions	–	

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 them 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.

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

Table 4 Comparison of the features of each requirement artifact

	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

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 from 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 helps 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.

Appendix: 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;
```

References

1. McDonald A, Welland R (2001) Web engineering in practice. In: Proceedings of the fourth WWW10 workshop on web engineering, pp 21–30
2. Lowe D (2003) Web system requirements: an overview. J Requir Eng 8(2):102–113. http://dx.doi.org/10.1007/s00766-002-0153-x
3. Jacobson I (1992) Object-oriented software engineering: a use case driven approach. ACM Press/Addison-Wesley, Boston
4. Rossi G, Schwabe D (2008) Modeling and implementing web applications using OOHDM. In: Rossi G, Pastor O, Schwabe D, Olsina L (Eds) Web engineering, modelling and implementing web applications, Springer, Heidelberg, pp 109–155
5. Valderas P, Pelechano V, Pastor O (2007) A transformational approach to produce Web applications prototypes from a Web requirements model. Int J Web Eng Technol IJWET 3(1):4–42
6. Koch N, Zhang G, Escalona MJ (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, Materna M (2003) Designing data-intensive web applications. Morgan Kaufman, Waltham
8. Gómez J, Cachero C (2003) OO-H method: extending UML to model web interfaces. In: van Bommel P (ed) Information modeling for internet applications. IGI Publishing, Hershey, pp 144–173
9. Escalona MJ, Koch N (2004) Requirements engineering for web applications–a comparative study. J Web Eng 2(3):193–212
10. De Troyer O, Casteleyn S (2003) Modeling complex processes for web applications using WSDM. In: Proceedings of the 3rd international workshop on web-oriented software technologies. Oviedo, Spain. At: http://www.dsic.upv.es/~west/iwwost03/articles.htm
11. Escalona MJ, Koch N (2006) Metamodeling requirements of web systems. In: Proceedings of the international conference on web information system and technologies (WEBIST 2006), INSTICC, 310–317, Setúbal, Portugal
12. Garrigós I, Mazón JN, Trujillo J (2009) A requirement analysis approach for using i* in web engineering. In: ICWE, LNCS, 5648, pp 151–165
13. Martin RC (2003) Agile software development: principles, patterns, and practices. Prentice Hall PTR, Upper Saddle River
14. Beck K (2002) Test driven development: by example. Addison-Wesley Signature Series
15. 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, pp 752–755. doi:http://doi.acm.org/10.1145/1101908.1101997
16. de Paula MG, da Silva BS, Barbosa SD (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 (2008) Web engineering: modelling and implementing web applications. Human-computer interaction series. Springer, London
18. GWT. Available at: http://code.google.com/webtoolkit/. Accessed 2011
19. Seaside. Available at: http://www.seaside.st/. Accessed 2011
20. Fowler M (2010) Domain specific languages, 1st edn. Addison-Wesley Professional, Boston
21. Claessen K, Hughes J (2000) 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
22. Bondy JA (1976) Graph theory with applications. Elsevier Science Ltd, Amsterdam
23. Balsamiq. Available at: http://www.balsamiq.com/products/mockups. Accessed 2011
24. Axure—wireframes, prototypes, specifications. Available at: http://www.axure.com/. Accessed 2011

25. Chomsky N (2003) Three models for the description of language. Inform Theory IRE Trans 2(3):113–124
26. Duhl J (2003) Rich internet applications. A white paper sponsored by Macromedia and Intel, IDC report
27. Yahoo patterns, http://developer.yahoo.com/ypatterns/
28. Moody D (2009) The physics of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans Softw Eng 35(6):756–779. doi:10.1109/TSE.2009.67
29. Maximilien EM, 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, pp 564–569
30. Robles Luna E, Grigera J, 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, Berlin, Heidelberg, pp 136–150
31. Eclipse EMF. Available at: http://www.eclipse.org/modeling/emf/. Accessed 2011
32. Eclipse GMF. Available at: http://www.eclipse.org/modeling/gmp/. Accessed 2011
33. Selenium web application testing system. Available at: http://seleniumhq.org/. Accessed 2011
34. jQuery: the write less, do more, JavaScript library. Available at: http://jquery.com/. Accessed 2011
35. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co, Boston
36. WebDriver. Available at: http://webdriver.googlecode.com. Accessed 2011
37. The WebRatio tool suite. Available at: http://www.webratio.com. Accessed 2011
38. Uden L, Valderas P, Pastor O (2008) An activity-theory-based model to analyse Web application requirements. Inform Res 13(2). http://informationr.net/ir/13-2/paper340.html
39. Conallen J (2000) Building web applications with UML. Addison-Wesley, Boston, p 300
40. Winckler M, Vanderdonct J (2005) Towards a user-centered design of web applications based on a task model. In: Proceedings of IWWOST'2005. Porto, Portugal, 12–13 June 2005
41. Flannagan S The paper version of the web. In: Deeplinking, available at: http://deeplinking.net/paper-web/
42. Lin J, Newman MW, Hong JI, Landay JA (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, 01–06 April 2000). CHI'2000. ACM, New York, NY, pp 510–517
43. Escalona MJ, Aragon G (2008) NDT. A model-driven approach for web requirements. IEEE Trans Softw Eng 34(3):370–390
44. Yu ESK (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, p 226
45. QVT. http://www.omg.org/spec/QVT/. Accessed 2011
46. Escalona MJ, Koch N (2006) Metamodeling requirements of web systems. In: Proceedings of the internacional conference on web information system and technologies (WEBIST 2006), INSTICC, pp 310–317, Setúbal, Portugal
47. Watir. Available at: http://watir.com/. Accessed 2011
48. Robles Luna E, Panach JI, Grigera J, Rossi G, Pastor O (2010) Incorporating usability requirements in a test/model-driven web engineering approach. J Web Eng (JWE) 9(2):132–156
49. Robles Luna E, Rossi G, Burella J, Grigera J (2010) Incremental usability improvement in an Agile approach for web applications. In: Proceedings of the 1st workshop dealing with usabiliy in an Agile domain, XP'2010 workshop, 2010, Trondheim, Norway
50. Robles Luna E, Garrigos I, Rossi G (2010) Capturing and validating personalization requirements in web applications. In: 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 (2010) An i*-based approach for modeling and tesing web requirements. J Web Eng (JWE) 9(4):302–326
52. Alencar FMR, Castro JFB (1999) Integrating early and late-phase requirements: a factory case study. In: Proceedings of XIII Brazilian symposium on software engineering—SBES99, Florianopólis, SC, Brasil, Outubro 1999, pp 47–61
53. Rivero JM, Rossi G, Grigera J, Burella J, Robles Luna E, Gordillo S (2010) From mockups to user interface models: an extensible model driven approach. In: Proceedings of the 6th model-driven web engineering workshop (MDWE 2010), Vienna, Austria