Dealing with Navigation and Interaction Requirements Changes in a TDD-Based Web Engineering Approach

Juan Burella[1,3], Gustavo Rossi[2,3], Esteban Robles Luna[2,3], and Julián Grigera[2]

[1] Departamento de Computación, Universidad de Buenos Aires
jburella@dc.uba.ar
[2] LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
[3] CONICET
{gustavo,esteban.robles,julian.grigera}@lifia.info.unlp.edu.ar

Abstract. Web applications are well suited to be developed with agile methods. However, as they tend to change too fast, special care must be put in change management, both to satisfy customers and reduce developers load. In this paper we discuss how we deal with navigation and interaction requirements changes in a novel test-driven development approach for Web applications. We present it by indicating how it resembles and differs from "conventional" TDD, and showing how changes can be treated as "first class" objects, allowing us to automate the application changes and also to adaptively prune the test suite.

1 Introduction

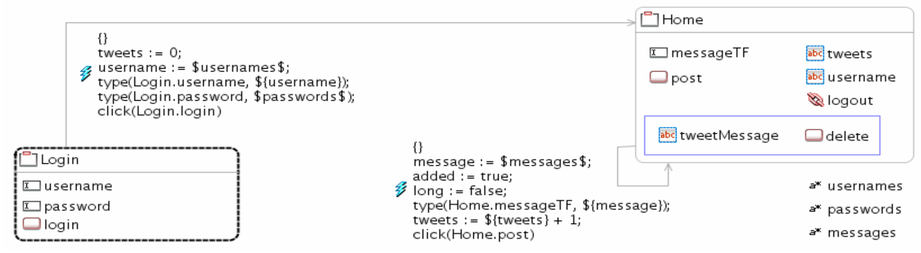
TDD and its variants like STDD [4] mostly focus on behavioural aspects of domain classes, and since TDD is generally applied in a bottom up way, it tends to disregard important features of Web applications such as navigation, interface or interaction. As a consequence, usability, look and feel, and also navigation features may be checked too late, once the application has been already presented to the customers, thus delaying the correction process.

As a way to overcome the mismatch between "conventional" TDD and Web applications development, we present an approach for improving change management in the context of our TDD-like mehotology by focusing on changes that affect navigation and interaction aspects. Requirements are represented using WebSpec diagrams, which capture navigation, user interface (UI) and interaction application aspects. WebSpec diagrams are then automatically translated into sets of meaningful interaction tests the application must pass. While the developers work coding the solution, the support environment captures the changes in objects and associates them to the corresponding tests. Change objects can also help to semi automatically change structural parts of the Web application when a requirement is added or changed. In the same way, we can reduce the number of tests that must be run to those that exercise changed objects only, improving the overall development time. We illustrate the approach with a simple Twitter-like application and show an integrated environment built on top of Seaside (www.seaside.st) that supports this functionality.

2 Background: A Test-Driven Approach for Web Applications

The key aspects of our requirements modelling stage are fast interface and interaction prototyping on one hand, and navigation modelling on the other. Prototyping is carried out with interaction mockups: simple HTML stub pages that significantly help to agree on the application's look and feel, and the way interaction must be performed.

We use WebSpec diagrams to specify navigation and interaction requirements more formally than with User Stories (US) [3]. These artefacts (based on UIDs [5] and Quickcheck [1]) capture navigation and interaction aspects in a similar way UIDs do, but adding the formal power of preconditions and invariants to assert properties in the interactions. A WebSpec diagram contains i*nteractions* and n*avigations*. An i*nteraction* represents a point where the user consumes information (expressed as a set of interface widgets) and interacts with the application by using some of its widgets. Some actions (clicking a button, adding some text in a text field, etc) might produce *navigation* from one *interaction* to another, and as a consequence, the user moves through the application's navigation space. These actions are written in an intuitive domain specific language. Fig. 1 shows a WebSpec that will let the user tweet, see how many tweets he has, and allow him to logout from the application. From the Login *interaction*, the user can authenticate by typing its username and password and then clicking on the login button (navigation from Login to Home interaction). Then, the user can add messages by typing in the messageTF and clicking on the post button (navigation from Home to Home interaction).



```
{}
tweets := 0;
username := $usernames$;
type(Login.username, ${username});
type(Login.password, $passwords$);
click(Login.login)
```

```
{}
message := $messages$;
added := true;
long := false;
type(Home.messageTF, ${message});
tweets := ${tweets} + 1;
click(Home.post)
```

Fig. 1. WebSpec of Tweet's interaction

From a WebSpec diagram we automatically generate a set of interaction tests that cover all the interaction paths specified in it, thus avoiding the translation problem of TDD between tests and requirements. Unlike unit tests, interaction tests simulate user input into HTML pages, and allow asserting conditions on the results of such interactions. Since each WebSpec *interaction* is related to a mockup, each test runs against it and the predicates are transformed into tests assertions. These (failing) series of tests set a good starting point for our TDD-like approach.

Once we have a set of tests for a specific requirement, it is time to develop the functionality to make them pass. Since interaction tests define the functionalities at user level, and we will drive the development from such tests, our approach will naturally follow a top-down style, rather than the usual bottom-up way of regular TDD. Nevertheless, we will use regular TDD as we dive into the application's underlying domain model.

Basing ourselves in the mockups, we recreate the same UI using widgets, but this time adding stubs for the dynamic behaviour in the places where the application will interact with the domain objects. Next, for every stub message left in the presentation and navigation classes, we code the model classes to fill the gaps.

At the time we engage in the development of the domain model classes, we follow a traditional TDD cycle by creating the unit tests first to establish the purpose of the new objects, which is in turn facilitated by the UI/navigation models which have already set specific requests for them. Once unit tests pass, we can run the interaction tests to check whether we have completed the necessary functionality for the current request. As usual, when tests do not pass, we keep working on the code until they do, and once this happens we can go on with the next requirement.

At times, some domain behaviour is needed, and it is not possible to state it as a UI requirement triggered by the user, or cannot be validated at the interaction level. In such cases, we capture the functionality using US and then create unit tests.

For the sake of conciseness, we will focus our explanation on navigation and presentation requirements, and therefore we will not talk about the effect of changes in "conventional" unit tests.

3 Our Approach to Change Management in a Nutshell

We borrow ideas from changeboxes [2] to make software changes explicit and manageable. We specifically focus on navigation and interaction changes in Web applications requirements to minimize the effort for satisfying their impact on the implementation. These changes are explicitly represented as first-class objects, and related to the artefacts in which they produce modifications, and as a consequence, we not only obtain better traceability features, but we are also able to automate some of these changes in the final application. Since navigation tests are also represented with objects, and we can determine the elements they access, and we can know exactly which tests are affected by a change. As a result, we can set apart the tests that will not check the new functionality at all, leaving only those that are really needed to check the new change and its consequences.

We have built a support tool that manages change objects and their relationships with the approach's artefacts. Being developed on top of the Seaside Squeak's environment (www.squeak.org), it allows to maintain these relationships during the whole life-cycle, helping to dynamically manipulate even application objects.

3.1 Representing Changes as First-Class Objects

As we show in Fig. 2, an application is developed by incrementally applying sets of changes (Step 4). Starting from the initial status, a first set of changes is applied in order to get an initial prototype. In further iterations, the application is extended with new sets of changes, fulfilling the requirements one by one. We stress that the main difference with "pure" TDD is that we automatically derive navigation and interaction tests from WebSpec diagrams, we actively use Mockups to derive the final application's interface, and we use interaction tests to guide the development of the application's behavior.

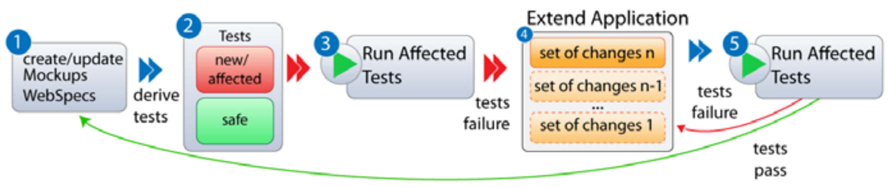


Fig. 2. A TDD-based Web Development Process

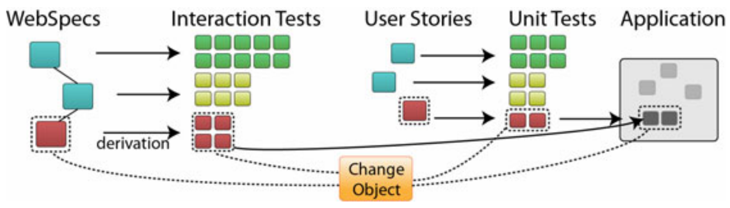


Fig. 3. Change Object's relationship with model and tests artefacts

Fig. 3 illustrates how the approach is improved using change management features. In the first stage, changes made in WebSpecs are captured into change objects. Then, changes made in the application's model are also captured into objects and associated with the corresponding test. We use these objects to reduce the set of interaction tests that drive the upcoming development steps to those affected by them.

WebSpecs can suffer different coarse grained changes, such as the addition or deletion of an *interaction* or *navigation*. These entities can be modified too, by the addition or deletion of widgets to an *interaction*, changes in invariants, etc. Regarding *navigations*, we can add or delete preconditions, change their source, target, or the action that triggers them. All these types of changes have been represented as classes.

3.2 Mapping Requirement Changes onto the Implementation

Some changes have direct effects on concrete application's artefacts; an important aspect of the corresponding change objects is that they can help to reduce the impact of these changes on the implementation. A *WebSpec* change object is associated to an effect on a Web artefact; this effect is also represented as a change object. These objects are able to produce the real modifications on their targets with the help of an *Effect Handler*. The *Effect Handler* is a component that knows how to perform changes on a concrete platform such as Seaside or GWT. For example, when a change modifies an interaction structurally, the page that represents this interaction must be modified: e.g. when a label is added to an interaction, it adds an equivalent label on the page represented by the modified interaction (Fig. 4).

Regarding *navigation* changes, we can change preconditions, sources, targets, or the actions that triggers them. The first type of change does not generate effects on the final application look and feel; in turn, if the *navigation* target changes, we can automate the effect of this change, for example linking the page associated to the new target *interaction*. Something similar happens when a *navigation* action is changed.

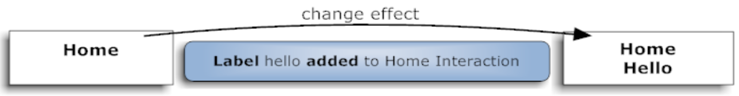


Fig. 4. A label addition change in action

4 A Proof of Concept

To illustrate our approach we describe how we put it into work in the Seaside framework, by implementing a specific *Effect Handler* for this platform. In the simplified Twitter-like application presented in Sect. 2, we started with a short sprint to capture the basic user stories: login and tweet. We will only discuss changes related with the tweet use story, and assume that we have finished the first iteration and the application satisfies the requirements captured by the WebSpec presented in Fig. 1.

Let us suppose that our customer wants to add the possibility to navigate from the home to a 'Terms of Service' page. In order to satisfy the new requirement, the development iteration starts with the requirements change (Step 1 in Fig. 2). We specify the *interaction* and *navigation* paths that we expect for the 'Term of service' requirement, which produces a set of change objects derived from the changes in the WebSpec diagram that express the link creation, the creation of the "terms of service" *interaction*, and the *navigation* between both *interactions* (Fig. 5).

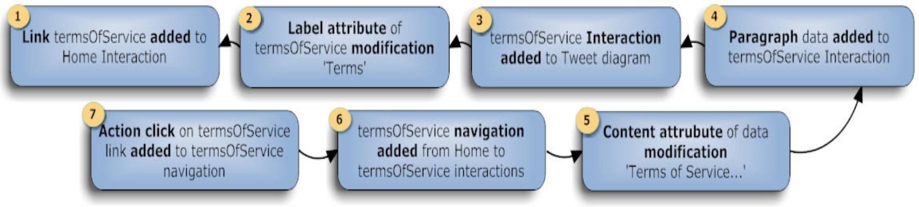

1 Link termsOfService added to Home Interaction
2 Label attribute of termsOfService modification 'Terms'
3 termsOfService Interaction added to Tweet diagram
4 Paragraph data added to termsOfService Interaction
5 Content attrubute of data modification 'Terms of Service...'
6 termsOfService navigation added from Home to termsOfService interactions
7 Action click on termsOfService link added to termsOfService navigation

Fig. 5. Change objects associated with the "Terms of Service" requirement

We then derive a set of tests from the WebSpec diagram and find that it generates a new test that checks the terms of service navigation, while in the previous iteration we had two tests that checked the addition of valid and invalid messages. To avoid running all three tests, we ask the Change Manager to determine which are affected by this change. As the only affected is the new one (Step 2 in Fig. 2) we run it, and it fails (Step 3 in Fig. 2), thus we must implement the new changes to satisfy this test.

The process continues with the change effect management (Step 4 in Fig.2) iterating over each change to see how it impacts on the implementation. The first change generates a creation method for the link widget in the *WAHome* class; it represents the page for Home interaction, so it will be drawn each time a *WAHome* instance shows. The next one modifies the widget label attribute to display the correct link name 'Terms'. Fig. 6 shows these effects. Change number 3 creates the Seaside component *WATermsOfService* that represents the page for this interaction. The next change generates a creation method for the paragraph widget in the *WATermsOfServices* class,

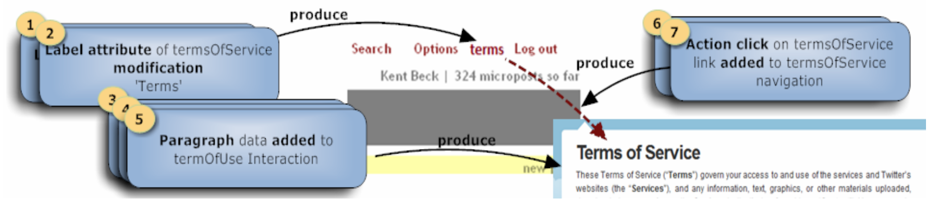

1
2
Label attribute of termsOfService modification 'Terms'
produce
Search Options terms Log out
Kent Beck | 324 microposts so far
6
7
Action click on termsOfService link added to termsOfService navigation
produce
3
4
5
Paragraph data added to termOfUse Interaction
produce
Terms of Service
These Terms of Service ("Terms") govern your access to and use of the services and Twitter's websites (the "Services"), and any information, text, graphics, or other materials uploaded,

Fig. 6. Managing the effects of the "Term of Use" requirement

and the next one modifies its content attribute, in the creation method. The *navigation* addition change does not produce modifications, but the last change generates the necessary code for associating the *interaction* pages through the "terms" link. Finally, we run the affected test realizing that it passes because of the semi automatic changes we applied on the application, thus completing the iteration (Step 5 in Fig.2).

5 Concluding Remarks and Further Work

In this paper we have presented an approach to deal with navigation and presentation requirement changes in the context of a TDD process for Web applications. Our main strategy has been to reify these changes into "first class" objects, so they can not only capture the history of changes, but also trace the effects of changes in different development artefacts, such as tests and application components.

An integrated tool built on top of the Squeak environment allows us to manipulate these change objects, making them extremely useful in the development process. In particular, we have shown how to help the developer by automating some modifications at the presentation level, or advising him about the necessary changes. At the same time, change objects allow reducing the number of tests that must be run, as they maintain a trace with their corresponding tests. Notice that this kind of change-aware development environment is easier to implement in a reflective system like Squeak, though much harder in Java-based environments such as Eclipse. In this sense, we are working on a light version of our environment for the Eclipse platform.

References

1. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: 5th ACM SIGPLAN international conference on Functional programming, pp. 268–279. ACM, New York (2000)
2. Denker, M., Gîrba, T., Lienhard, A., Nierstrasz, O., Renggli, L., Zumkehr, P.: Encapsulating and exploiting change with changeboxes. In: 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007, vol. 286, pp. 25–49. ACM, New York (2007)
3. Jeffries, R.: Extreme programming installed. Addison-Wesley, Boston (2001)
4. Mugridge, R.: Managing Agile Project Requirements with Storytest-Driven Development. IEEE software 25, 68–75 (2008)
5. Rossi, G., Schwabe, D.: Modeling and Implementing Web Applications using OOHDM. In: Web Engineering, Modelling and Implementing Web Applications, pp. 109–155. Springer, Heidelberg (2008)