Bridging Test and Model-Driven Approaches in Web Engineering

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

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

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

1 Introduction

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

Unfortunately most solid Model-Driven Web Engineering (MDWE) approaches, even claiming to favor incremental and iterative development, use a more formal[1] and waterfall style of development. Web engineering methods like UWE [14], WebML [6], OOWS [18], OO-H [9] or OOHDM [22] define a set of abstract models such as content (called also data or application), navigation and presentation model, which allow the generation of running applications by automatic (error free) model transformations. This approach is attractive because it raises the abstraction level of the construction process, allowing developers to focus on conceptual models instead of code. The growing availability of techniques and tools in the universe of model-driven development (e.g. transformation tools) adds synergy to the approach.

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

M. Gaedke, M. Grossniklaus, and O. Díaz (Eds.): ICWE 2009, LNCS 5648, pp. 136 – 150, 2009.

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

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

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

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

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

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

The main contributions of the paper are the following:

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

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

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

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

2 Related Work and Discussion

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

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

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

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

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

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

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

3 An Overview of Our Approach

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

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

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

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

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

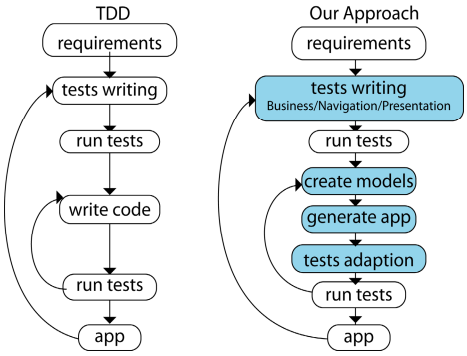


Fig. 1. TDD life cycle comparison

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

3.1 Capturing Requirements with Mockups and UIDs

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

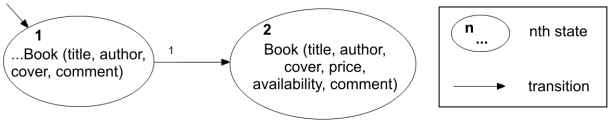
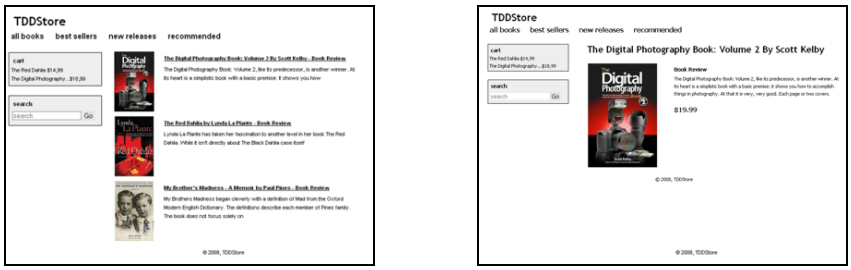


Fig. 2. UID for simple navigation

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

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



(a) (b)

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

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

3.2 Writing Tests

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

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

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

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

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

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

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

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

3.3 Deriving Design Models

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

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

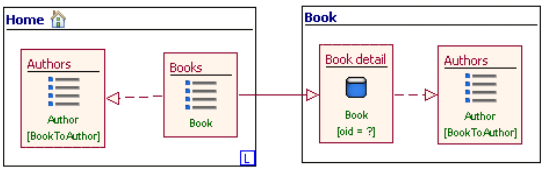


Fig. 4. WebML diagram for the UID

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

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

3.4 Test Adaptation

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

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

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

```
public class BookListTestCase extends SeleneseTestCase {
   public void testBookListToProdDetailNav() throws Exception {
(1)  sel.open("http://127.0.0.1:8180/TDDStore/page1.do");
(2)  assertEquals("all…", sel.getText("//div[@id='page1FB']/p[1]"));
(3)  sel.click("link=The Digital Photography Book");
```

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

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

3.5 Towards a New Iteration

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

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

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

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

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

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

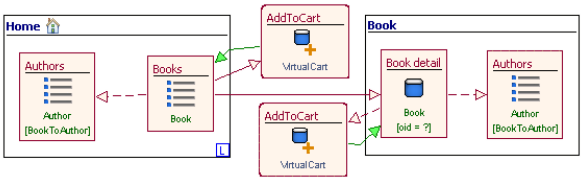


Fig. 5. Upgraded WebML diagram

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

4 Dealing with Application Evolution

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

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

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

4.1 New Requirements

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

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

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

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

4.2 Web Refactorings

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

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

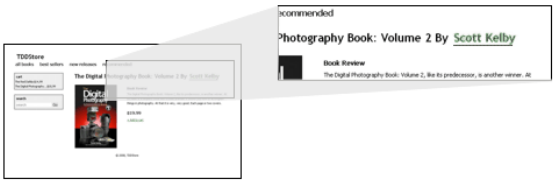


Fig. 6. Refactored book details mockup

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

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

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

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

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

5 Towards Automated Test Evolution

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

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

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

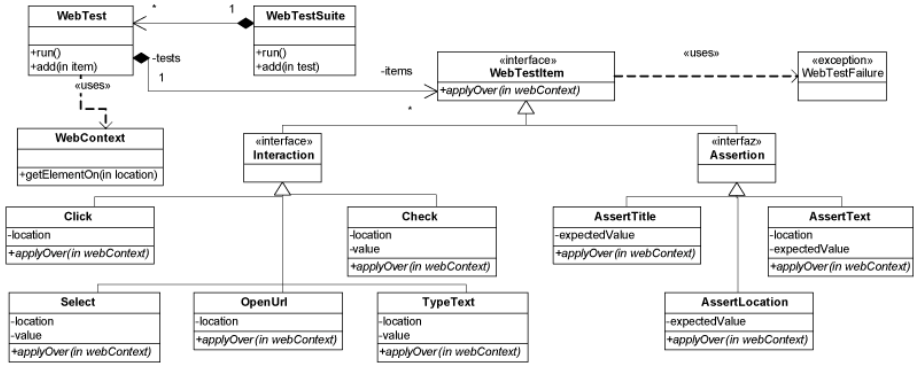


Fig. 7. Web Test Model

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

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

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

6 Concluding Remarks and Further Work

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

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

References

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

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