From Requirements to Web Applications in an Agile Model-Driven Approach

Julián Grigera[1], José Matías Rivero[1,2], Esteban Robles Luna[1], Franco Giacosa[1], and Gustavo Rossi[1,2]

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

Abstract. Web applications are hard to build not only because of technical reasons but also because they involve many different kinds of stakeholders. Involving customers in the development process is a must, not only while eliciting requirements but also considering that requirements change fast and they must be validated continuously. However, while model-driven approaches represent a step forward to reduce development time and work at a higher level of abstraction, most of them practically ignore stakeholders' involvement. Agile approaches tend to solve this problem, though they generally focus on programming rather than modeling. In this paper we present an extension to an approach that combines the best of both worlds, allowing a formal and high-level design style with constant involvement of customers, mainly in the definition of navigation, interaction and interface features. We extended it by adding transformation features that allow mapping requirement models into content and navigation ones. We provide a proof of concept in the context of the WebML design method and an empiric validation of the approach's advantages.

1 Introduction

Developing Web applications is a complex task, involving different specialists through different stages. At the end of the process, it is usual to find out that the final result does not reflect the customers' wishes with accuracy, since while going through the different stages the team may slowly steer away from the original requirements. The difference between requirements and the final result grows broader as new changes are introduced. These problems are in part caused by communication issues, but they also arise as a consequence of the development approach.

In a previous work [17] we argued that most model-driven Web engineering approaches (MDWE) [1, 8, 12, 19] tend to focus on the design artifacts and their automatic transformation onto running applications, therefore leaving the customer aside (at least in part) throughout the process. Interaction and interface issues are usually left as final concerns, while being, in many applications, the most important aspects for customers. At the same time agile approaches[1] focus on customers' involvement,

[1] Principles behind the Agile Manifesto – http://agilemanifesto.org/principles.html

M. Brambilla, T. Tokuda, and R. Tolksdorf (Eds.): ICWE 2012, LNCS 7387, pp. 200–214, 2012.
© Springer-Verlag Berlin Heidelberg 2012

while being less *formal* from the technical point of view. We then proposed to bridge both approaches by using Test-Driven Development (TDD) in a model-driven setting. With short development cycles, the mismatch between requirements and implementation is usually kept under control. We already proposed a requirement engineering language, named WebSpec [18] to capture navigation and interaction requirement. Associated with customer-generated mockups, WebSpec diagrams provide simulations to share an early view of the application with stakeholders and automatically derive acceptance tests (using test frameworks like Selenium[2]).

In this paper we go one step further from these two previous contributions by showing how to semi-automatically derive navigation and domain models from requirements captured with mockups and WebSpec diagrams. Interface mockups are not thrown away as usual (even in agile approaches) but evolve into the final applications' interface. The approach, which incorporates requirements into the model-driven cycle, is still agile in that it is based in short cycles with heavy customers intervention, since the used requirements artifacts (Webspec diagrams and mockups) can be manipulated by them; however it can also be used in a conventional "unified" model-driven style.

Though the approach is agnostic to the underlying design method, we illustrate it with the WebML [1] notation and its associated tool WebRatio[3] with which we have made extensive experiments. We also show that the approach does not necessarily depend on interaction tests as driving artifacts for the development (like most TDD approaches do); therefore it can be used either with organized agile styles like Scrum, or even with more "extreme" approaches [6].

The main contributions of the paper are: first, from a process point of view, a way of bridging agile and MDWE from requirements to implementation, easing customer participation from early stages of development using interface mockups and fast prototype generation as a common language to discus requirements; second, we provide a shorter path from requirements to models through a set of heuristics to transform requirement models (expressed as WebSpec diagrams plus interface mockups) onto navigation, presentation and content models. We illustrate these contributions with a set of running examples and describe an experiment that validates our claims.

The rest of the paper is structured as follows: in Section 2 we present a brief background of our work emphasizing on WebSpec diagrams and interface mockup annotations. Next, in Section 3 we explain our approach in detail. In section 4 we show a simple but meaningful example. Section 5 shows an experiment that validates the approach and Section 6 presents some related work on this subject. Section 7 concludes the paper and discusses some further work we are pursuing.

2 Background

The first stage of our process involves two main artifacts that help to state clearly what customers need, and how they want it to look and behave. Graphical user interface (GUI) mockups combined with WebSpec diagrams will not only help through this stage, but also in the following, as we will explain later on section 4. Besides these artifacts, we will organize the requirements gathering with User Stories [6] as functional units, though Use Cases [5] can also be used for the same purpose.

---

[2] Selenium web application testing system - `http://seleniumhq.org/`

[3] WebRatio – `http://www.webratio.com`

2.1 GUI Mockups

GUI mockups serve well as first requirement artifacts, since they are really close to customers in terms of interfaces and interaction, resulting much clearer than textual specifications. Mockups act as tools to communicate software requirements in a *common language* shared between customers and the development team [10]. It has been shown that screen mockups effectively increase general software comprehension without involving a high cost in the development process [15]. Besides, we have shown that they also work as specifications for building user interface models [16]. When built using digital tools, mockups represent an incomplete, yet non-ambiguous, description of the UI. However, in most cases mockups are used only during the requirements specification and thrown away shortly after. We have also shown that, because of the common fidelity (i.e., the shared abstraction level and metamodel elements) between MDWE presentation models and modern mockup building tools, we can easily translate mockups to UI models using a transformation process [16].

In this work we employ user interface mockups as the initial artifacts to interact with customers. Once agreed upon them, mockups are derived into the presentation model of the application (that we can generate automatically) and a foundation to specify further features, like navigation and content aspects.

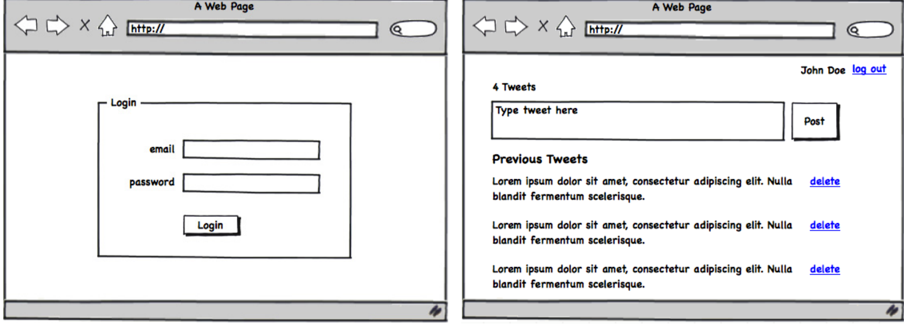


Fig. 1. Sample GUI mockups

Figure 1 shows two simple low-fi mockups of a login and home screen to a Twitter-like application. Later in the paper we show how they can be combined with a WebSpec diagram to describe the navigation features these artifacts lack.

2.2 WebSpec

WebSpec [18] is a DSL designed to capture navigation and interaction aspects at the requirements stage of a Web applications development process. A WebSpec diagram contains *Interactions* and *Navigations*. An Interaction represents a point where the user consumes information (expressed as a set of interface widgets) and interacts with the application by using its widgets. Some actions like clicking a button, or typing some text in a text field 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. Figure 2 shows a diagram that will let the user tweet, see how many tweets she has, and allow her to logout from the application. From the Login interaction, the user types username and

password and clicks on the login button (navigation from Login to Home interaction). Then, she can add messages by typing in the message text field (messageTF attribute) and clicking on the post button (navigation from Home to Home interaction).

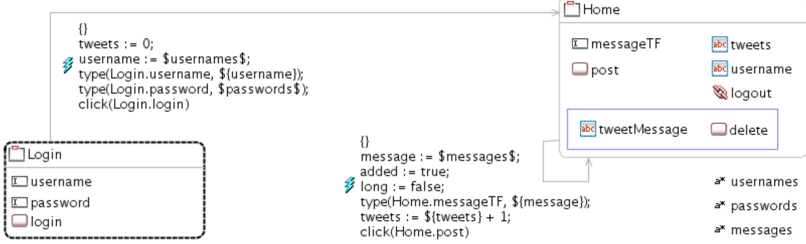


Fig. 2. WebSpec of Tweet's interaction

From a WebSpec diagram we automatically generate a set of interaction tests that cover all the interaction paths specified in it [18], avoiding the translation problem of TDD between tests and requirements. Unlike traditional 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 a TDD-like approach and (even when using another agile approach) they can be used later as the application's acceptance tests.

3 The Approach in a Nutshell

To bridge the gap between requirements specifications and implementation, we have devised model transformation rules for turning requirements artifacts into content and navigation models. We depict the approach in Figure 3 assuming a TDD cycle.

The process begins with a small group of initial requirements, related to a single User Story. We gather presentation and interaction requirements by building interface mockups, which help to agree upon the look and feel of the new application, and will also provide the basis for WebSpec diagrams.

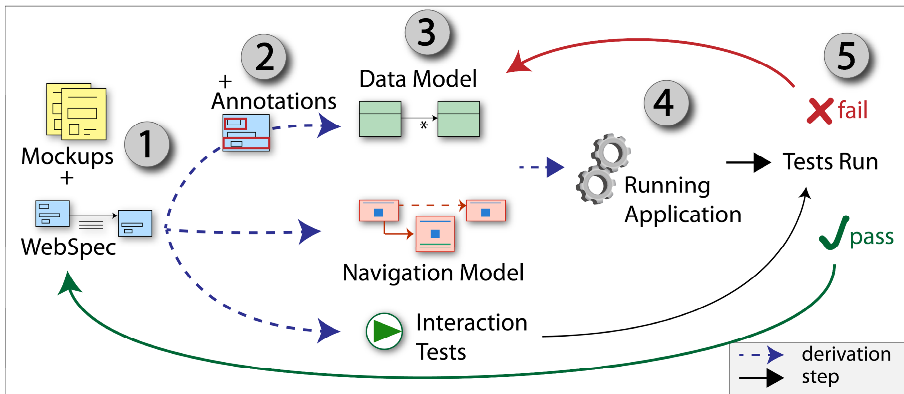


Fig. 3. Summary of the approach

After building the mockups, we specify navigation features through WebSpec diagrams. Since WebSpec can express interaction requirements (including navigation), general hypertext specifications can be derived directly from it, but backend features are missed, being the most important the underlying content model. To fulfill this gap, we annotate WebSpec widgets to represent content model features, in terms of classes (or entities) and attributes. These annotations are extremely simple and easy to apply and will help to build the content model incrementally and in an *on-demand* fashion.

Once we have both mockups and the annotated WebSpec diagrams, we derive a first set of content and navigation models. We generate the navigation model from the WebSpec diagrams directly, and we make use of the annotations made on them to derive the content model. Both models are linked together automatically since they stem from the same diagrams. Additionally, WebSpec diagrams are used to generate the interaction tests [18] that will guide the rest of the development in an agile style.

Having created the models with their corresponding interaction tests, the developers apply the presentation according the mockups devised in the first stage and derive a running application, which must be validated with such interaction tests. When using a TDD style, if tests fail, the models must be tweaked until they pass, and then move forward to another User Story for the following iteration towards the final application. The reason why interaction tests might fail is because the transformation rules can sometimes be inaccurate, and while these misinterpretations are mostly due to insufficient information in the WebSpec diagrams or their annotations, some can also be due to ambiguous customer's specifications. In such cases, the type of corrections required to adjust the models to their correct semantics have proven to be recurrent, so we devised a list of frequent model adjustments in a pattern-like style.

In the following subsections we detail how we specify WebSpec diagrams, then turn them into navigation and content models through a set of transformation rules, and the main required refactorings we detected for correcting the derived models.

3.1 Gathering Navigation Requirements with WebSpec Diagrams

We use the existing tooling support for WebSpec to import and group existing mockups as defined in the initial User Stories. For every mockup in each User Story, a WebSpec Interaction is created to specify the behavior mockups cannot express. With assistance of the tool, mockup widgets can be *projected* to WebSpec diagrams in order to be included in interaction specifications.

It is important to note that a single mockup can be referenced by two or more WebSpec Interactions in different diagrams, since many User Stories can be partially

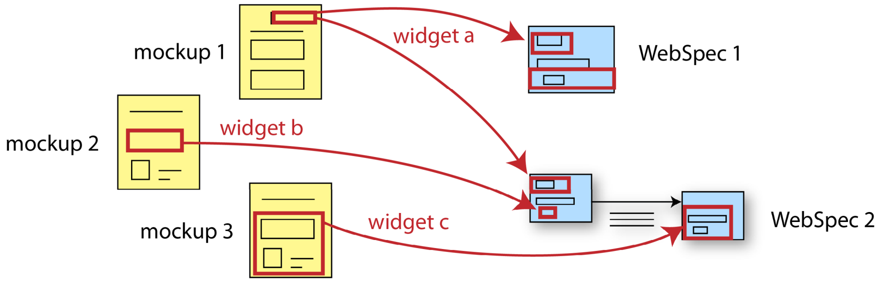


Fig. 4. Overlapping mockups and WebSpec diagrams

related to the same user interface in the Web application. Conversely, two or more mockups can be referenced in a single WebSpec diagram, given it specifies navigation from one to another. This is shown schematically in Figure 4.

3.2 Obtaining Data Model through Annotations

After creating the WebSpec diagrams, we apply lightweight content annotations on their widgets (for a complete reference on widgets see our previous work on WebSpec [18]); this will allow us to generate content models *on the fly* together with interaction specs. Generating content models from structured UIs have been already proposed and implemented [14], here we define an extremely simple annotation schema that can be applicable directly with the annotation facilities provided with mockup tools:

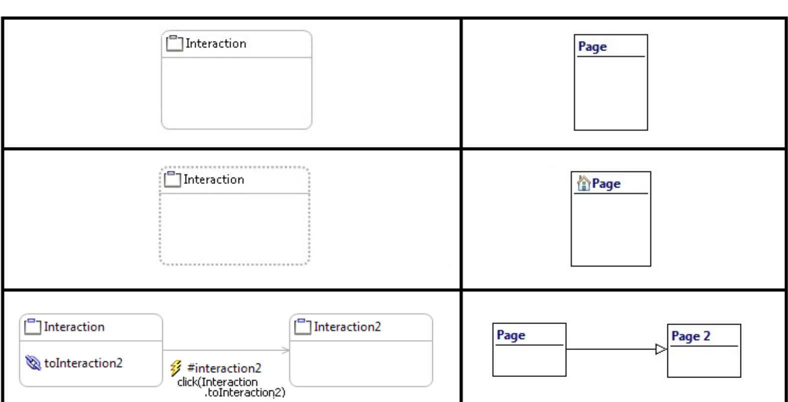
- Composite widgets (Panels and ListPanels) are annotated with a single string that denotes the class (or entity) it handles (e.g., `@Employee` in Figure 5.a).
- Simple input widgets (like TextFields or Checkboxes) are annotated with the syntax `<class>.<attributeName>` (`@Employee.Role` in Figure 5.b), also applied to simple widgets referring other classes' instances (like ComboBoxes or Lists).

Fig. 5. Annotated WebSpec diagrams

3.3 Deriving Models

In this section we show how we obtain navigation and data models from WebSpec diagrams. We begin with some basic transformations that intuitively map simple WebSpec constructions into WebML elements, shown in table 1.

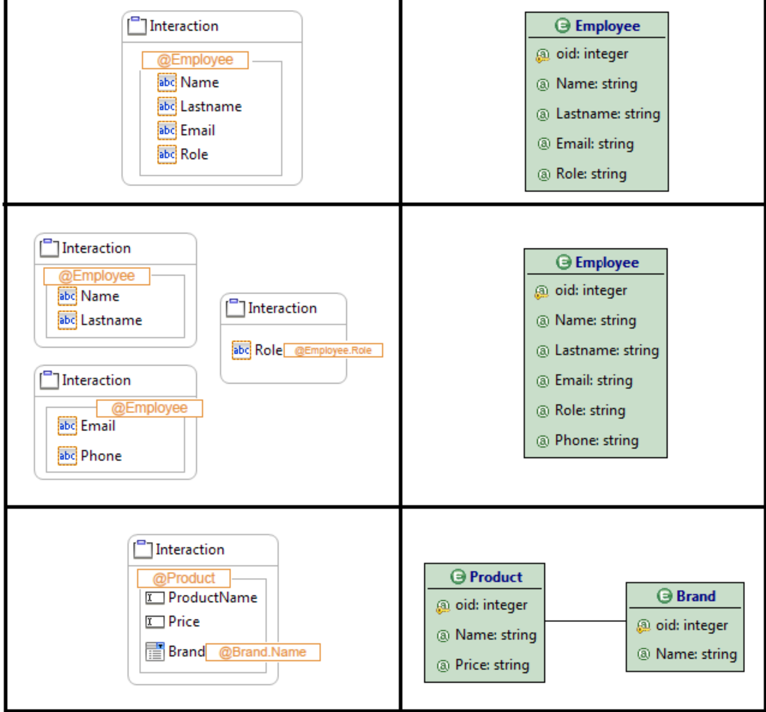
Table 1. Basic WebSpec to WebML transformations



The first transformation rule maps a Webspec Interaction to a WebML Page. Every WebSpec diagram is initialized with a Starting Interaction component that will be represented using a WebML Home Page. A link between Interactions will be turned into a Normal Link in WebML.

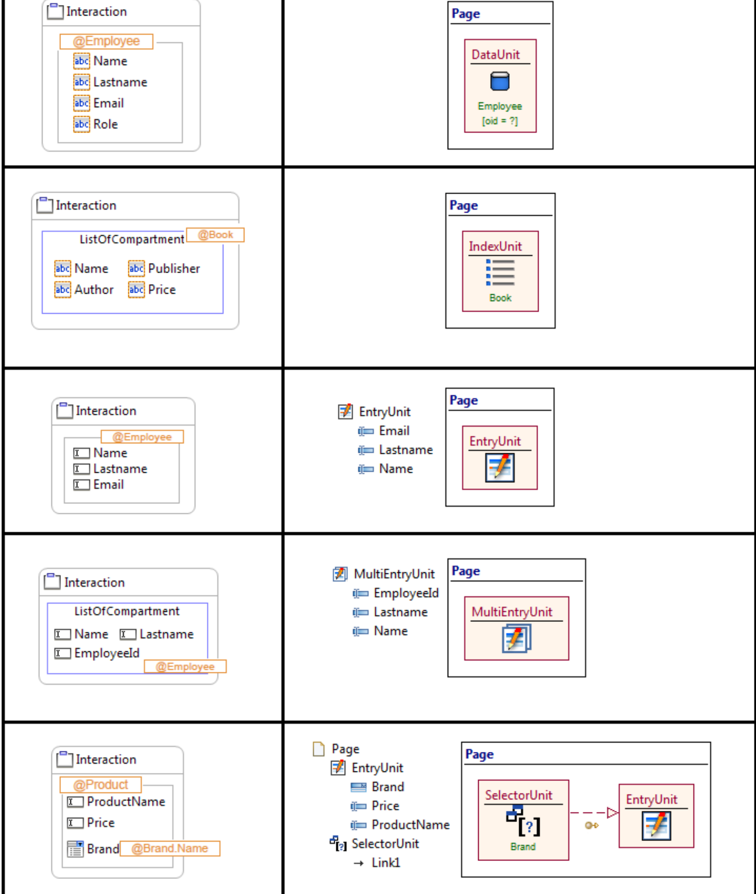
The annotation schema (explained in section 3.2) combined with the WebSpec model allows us to derive a WebML Data Model as well. In table 2 we depict some transformation rules including content model annotations.

Table 2. From annotated WebSpec to Data Models



The `@Class` annotation (e.g. `@Employee`) allows specifying that the underlying composite widget will manage instances of the `Class` entity. As a consequence, a corresponding WebML Entity will be created in the WebML Data Model and every simple widget in it will be transformed as an attribute (the OID attribute will be added by default to each new entity). If an entity is spread in several diagrams, a union operation will be applied to create the entity. Each simple widget found either by been inside a Composite Widget annotated with `@Class` or by being annotated with the `@Class.attribute` label, will be gathered an put inside a single Entity as long as they share the same Class. If a Simple Widget inside a Composite Widget has a different class annotation than its parent, a relationship between the class of the Composite Widget and the one of the Simple Widget will be created. After deriving a Data Model, now we can start mapping the above WebML Web Model, as we show in the transformations portrayed in Table 3.

Table 3. Obtaining full WebML models



The transformations introduced in Table 3 are the following: (1) A Panel used to show data (e.g a panel of labels) with a `@Class` annotation is transformed to a Data Unit pointing at the specified class, (2) A List used to show data (e.g. a list of labels) with a `@Class` annotation is transformed to an Index Unit pointing at the specified class, (3) A Panel used to input data (e.g. a panel composed by input widgets) with a `@Class` annotation is transformed to an Entry Unit and each input widget in the panel will be mapped to a WebML input field, (4) A List used to input data (e.g. a list composed by input widgets) with a `@Class` annotation is transformed to an Multi Entry Unit and each input widget in the List will be mapped to a WebML input field and (5) if inside an Input Panel there is a Combo Box annotated with a different class (e.g.: `@Brand.name`), a selection field will be created in the Entry Unit, and it will be filled with a Selector Unit pointing at the specified entity annotated in the Combo Box.

Before the derived models are ready to generate a running application, some fixes might need to be made. We will discuss these in the next section.

3.4 Adjusting the Models

Applying the described transformations to the initial requirement artifacts, both navigation and data models are generated in conjunction with a set of interaction tests, as depicted in Figure 3. Using the code generation capabilities of the chosen MDWE approach, a running application is generated in order to run the interaction tests over it to check the functionality. In some cases, tests may fail on their first run, due to missing or unexpected presentation details or layout specifications in the final user interface. However, in some cases they can also fail because of ambiguous or insufficient behavior inferred from the models derived with the described rules. Regarding data and business logic, we found a list of *fail patterns* and devised some heuristics to detect them and suggest potential corrections. Depending of its importance and obviousness, fail patterns are presented to the designer as a refactoring [4] suggestion in the tool or they are applied automatically as a final part of the generation process. We detail two notorious examples below:

- **Non-normalized Attribute**
  - Explanation: a simple widget is bound to an individual attribute of a mapped class, but in fact it must be bound to an attribute of a different class related to the former through an association.
  - Example: a product panel tagged as `@Product` has a label called `brandName`. This label must not be data-bound as an attribute of `Product`, but to an attribute of `Brand`, a class associated to `Product`. Then, a proper `@Brand.name` annotation must be placed in it (see Figure 6).
  - Fail reason: data is not normalized and fails occur when updating information within the execution of a WebSpec test (e.g., the brand name of a product is changed, and when the test checks the name in a second product of the same brand, it has the old one and an equality assertion fails).
  - Detection Heuristic: analyze widget name and search for the name of a previously mapped class within it. Suggest an association to this class.
- **Missing Filter in Index**
  - Explanation: an input panel and a list exist in an interaction. The panel contains widgets that specify filtering conditions to the elements that are shown on the list. Both widgets are annotated with the same class and a transition from the interaction to itself exist. According to the translation rules, a WebML Index Unit will be generated in the model for the list and an Entry Unit must be created for the input panel. However, no filtering is generated by default.
  - Example: an interaction contains a panel with a textbox that allows searching products by its name. Below, a list of products found with the matching name is shown (see Figure 7).
  - Fail reason: items in the index are the same after changing the *filter widgets* values in the panel and updating the page. Thus, an equality assertion fails.
  - Detection Heuristic: analyze the interaction to find a panel and a list annotated with the same class and a recursive transition.

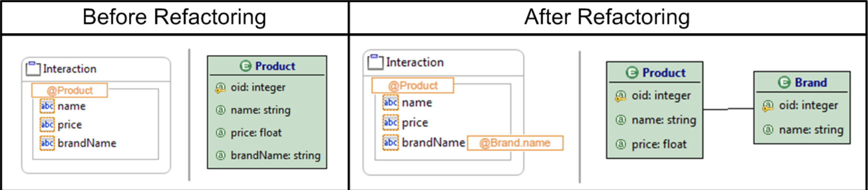


Fig. 6. Non-Normalized fail pattern and refactored diagram

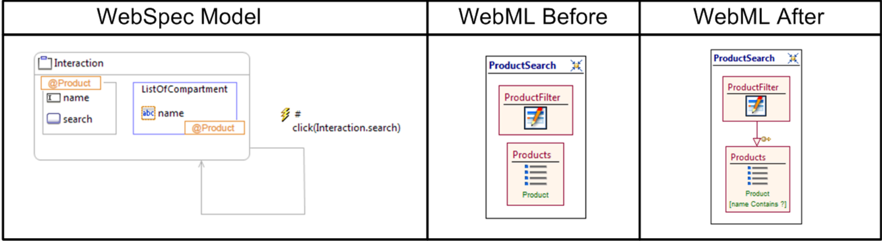


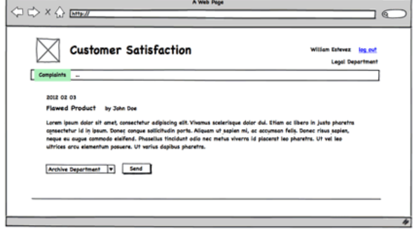
Fig. 7. Missing *Filter in Index* fail pattern and refactored diagram

4 Proof of Concept

For a better understanding of the approach, we will show a full cycle of our process in the ongoing development of a sample application: a Customer Satisfaction system, where different users manage customers' complaints through different departments.

We will take the development from an advanced status, and show how a new User Story is implemented. We will start from a point where the system allows creating new complaints, viewing their details and delegating them between departments. The next functionality to implement is the ability to make comment on the complaints.

As a first step, the previous mockup for the detailed view of a complaint is extended to show comments and a new form is added for the user to leave comments.



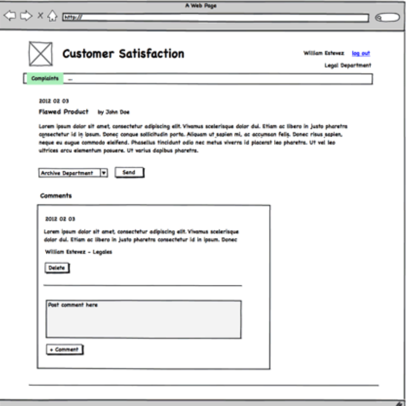


Fig. 8. Mockups for new functionality

Figure 8 shows the previous mockup for the details page of a complaint, and the new mockup that contemplates the comments.

Once we have agreed on the new functionality's look and feel, we move on to the WebSpec diagrams. Since the interaction for viewing a complaint was already present, we just extend it with the list of comments and the form for adding comments, with the corresponding navigation functionality. Figure 9 shows both previous and modified diagrams.

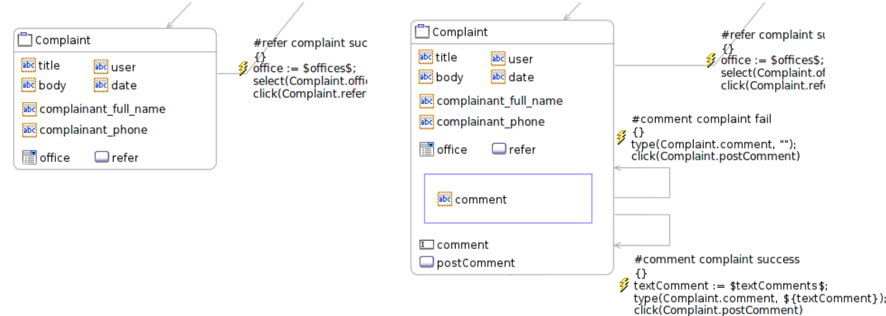


Fig. 9. Extended WebSpec model for comments feature

We next tag the new components of the diagram with annotations for deriving the missing content model. Then, the only step left is derivation. The extended WebSpec diagram generates new features for the existing navigation model, while the annotations generate a simple model for the comments, and their relationships with the complaints on the current data model. New interaction tests are also generated to check for the creation of new comments.

As a last step we regenerate the application from the derived models, and run the automatically generated interaction tests to validate the new functionality. If the tests pass, we move on to another User Story; if they don't, we must check for possible inaccurate derivations. For example, in this case we could have specified the author's name for the comments as a plain attribute for the Comment entity, instead of being a foreign attribute from the User entity, which should be related to the first (*Non-normalized Attribute* fail pattern). Fixing the data model will require also fixing the navigation model, and re-running the tests to check for the functionality.

5 Assessing the Approach

To make a first assessment of our approach we ran an experiment with 10 developers, each going through a complete development cycle for a simple application: the Complaint Management System presented as example in section 3.

We split the subjects in 2 groups of 5 developers, each group using different approaches in the requirements elicitation and WebRatio as development tool. A first group (group A) used only User Stories and UI Mockups, while the second one (group B) added also WebSpec and tagging, completing the full approach proposed in this paper, relying on the models derivation features.

We had a first meeting with each subject playing the role of customers. Depending on the group they belonged, they gathered requirements using different artifacts. They

were also provided with ready-made User Stories. Before the second stage, the developers from group B automatically derived content and navigation models from the WebSpec diagrams they had created and tagged. Then, all subjects developed the complete application measuring the time taken to implement each User Story individually. Additionally, developers from group B measured the time taken to alter the derived models to make up for eventual derivation mistakes. The third stage involved acceptance tests to check all functionality in both groups' resulting applications.

In this experiment we measured two key aspects: time and satisfaction. The latter measures the functionality's accuracy to what users expected, in a scale ranging from 1 to 5. We found an improvement in both aspects. In figure 10 we depict the average time in minutes it took for each group for completing each User Story.

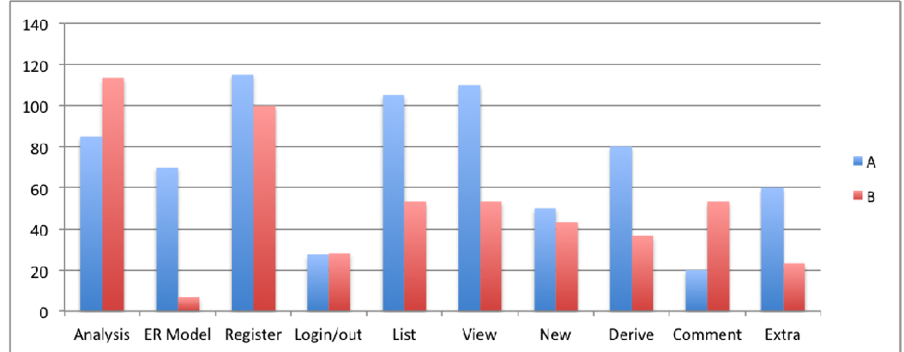


Fig. 10. Time measured for development

As the graphic shows, the time taken by the group A was considerably higher than group B's, mostly owed to the fact that group B only adjusted models, while group A had to create them from scratch, including the data model (marked as ER in the chart). Also, group A took more time to develop extra features that were not asked for in the requirements documents. However, Group B took longer to capture requirements, since they had more artifacts to put together.

As for the satisfaction aspect, the results were not as much conclusive as the previous ones, although they did show an improvement in most User Stories. After

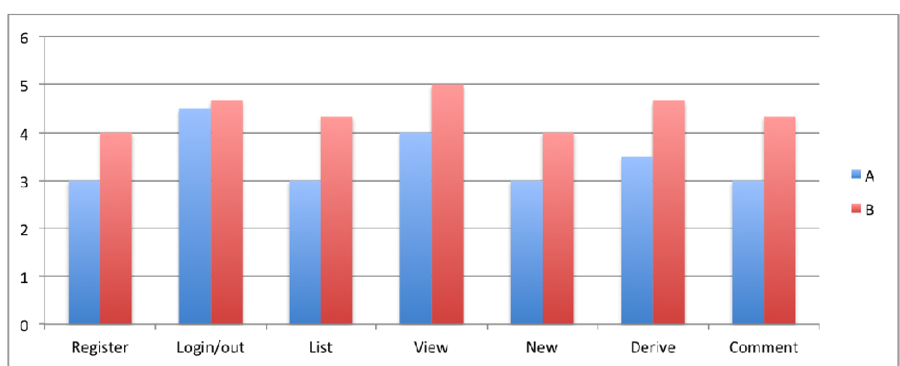


Fig. 11. Satisfaction measured for development

revising the applications we concluded that using WebSpec did improve the fidelity of the final application with respect to the requirements elicitation. Figure 11 shows the average satisfaction ratings for all functionalities developed by both groups.

It should be mentioned that the validity of these first results was somewhat threatened by a number of biases, mainly due to time and resources limitations. Some of them were: the application's scale (which is by no means a real scale development), the small number of subjects and the difference in their skills. The novelty in the use of WebSpec was also a factor, manifested in the higher analysis times on Group B. We plan to make further experiments with more experienced subjects on the use of WebSpec and it's tools, to confirm the presumption that analysis times will drop, at least to the levels of a traditional approach.

6 Related Work

Derivation of requirement models has been already considered with the aim of automatically generating UWE models [7]. In this work, the authors present a modeling language for requirements called WebRE, using the NDT approach [2] for the requirements capture and definition, and specify a set of transformation rules, specified at the meta-model level in the QVT language. The transformation process covers the derivation of content, navigation and presentation models.

Following the same lead, the Ariadne CASE tool [9] generates design models from requirement models, in the context of the ADM model-driven approach, used in turn to generate light prototypes of the final application. The tool leans on domain-specific patterns for generating conceptual models.

Also in this field, Valderas et al [21] propose an improvement on their automatic code generation from OOWS, in which they include graphic designers into the development. To do this, they automatically extract information and functionality from the requirements models. This allows the designers to make changes on a living application for a better experience in the requirements gathering stage, but the presentations are not part of the requirements models from which the information and functionality is extracted.

Our process differs from the aforementioned approaches in that it is focused on short agile development cycles. Being based on GUI Mockups and WebSpec, which is in turn based on User Stories, we not only favor an agile style, but also are able to generate interaction tests to check the resulting applications, in a way that lets us take advantage of the features of TDD approaches as well.

With respect to the artifacts used in our approach, GUI Mockups as requirements gathering tools have been evaluated in several studies. In the context of agile development processes, interface mockups have been observed as an irreplaceable artifact to effectively introduce early usability testing [3]. Also, they have proven to help refining concepts expressed in User Stories [20].

On the other hand, user interface mockups have been included in well known Model-Driven methodologies to improve requirements gathering. In the work of Panach et al. [11], the drawing of user interface sketches is proposed as a way of capturing underlying task patterns using the ConcurTaskTree [13] formalism. Other authors propose directly to include mockups as a metamodel itself to describe interaction from them [10].

7 Concluding Remarks and Further Work

Through this paper we have shown how we improved an agile model-driven Web development process by including digital requirement artifacts to keep the whole process model-driven; in this way we bridge the gap between requirements and implementation by introducing model transformations that automatically map requirement models into content and navigation models; this models are ready to be used to generate a running application, which can be in turn validated using the automatically generated interaction tests.

By driving an experiment with developers, we have shown the strengths of the approach, concerning not only requirements gathering stage but also the rest of the development process. The experiment has also exposed some weaknesses in the derivation process as well as in the process itself, on which we are already working to improve, before running a new, more comprehensive, experiment. In the same way we discovered the current transformation rules, we noticed that the combination of data annotations and navigation features of WebSpec models has still potential for new transformation rules that require further experimentation in order to be correctly stated. We are also finishing derivation rules for object-oriented approaches like UWE [8], which are resulting straightforward since we are working at the meta-model levels of WebSpec and UWE. At the same time we are also extending the WebSpec meta-model to introduce new requirement features.

Regarding the model adjustments, we are working on a suggestion mechanism that will be integrated into our tool, in order to detect possible miscarried derivations and correct them automatically, prompting a set of applicable corrections to the user for him to pick the most suitable one.

Another concern we are working on is the relationship between requirements and implementation models after the transformations. In order to keep track of such relationship and being able to generate changes incrementally, at this point we do not allow for mayor modifications on the application's models. The only modifications allowed should be those that do not introduce changes in the requirements – i.e. what WebSpec diagrams express. Nevertheless, we intend to handle these cases in such a way that allows us to suggest changes on the WebSpec diagrams, so the link between them and the generated models is never broken.

References

1. 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)
2. Escalona, M.J., Aragón, G.: NDT. A Model-Driven Approach for Web Requirements. IEEE Trans. Softw. Eng. 34(3), 377–390 (2008)
3. Ferreira, J., Noble, J., Biddle, R.: Agile Development Iterations and UI Design. In: AGILE 2007 Conference (2007)
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
5. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. ACM Press/Addison-Wesley (1992)

6. Jeffries, R.E., Anderson, A., Hendrickson, C.: Extreme Programming Installed. Addison-Wesley Longman Publishing Co., Inc. (2000)
7. Koch, N., Zhang, G., Escalona, M.J.: Model transformations from requirements to web system design. In: Proceedings of the 6th International Conference on Web Engineering (ICWE 2006), pp. 281–288. ACM, New York (2006)
8. 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)
9. Montero, S., Díaz, P., Aedo, I.: From requirements to implementations: a model-driven approach for web development. European Journal of Information Systems 16(4), 407–419 (2007)
10. Mukasa, K.S., Kaindl, H.: An Integration of Requirements and User Interface Specifications. In: 6th IEEE International Requirements Engineering Conference (2008)
11. Panach, J.I., España, S., Pederiva, I., Pastor, O.: Capturing Interaction Requirements in a Model Transformation Technology Based on MDA. Journal of Universal Computer Science 14(9), 1480–1495
12. Pastor, Ó., Abrahão, S., Fons, J.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)
13. Paternò, F.: ConcurTaskTrees: An Engineered Notation for Task Models. In: Diaper, D., Stanton, N. (eds.) The Handbook of Task Analysis for Human-Computer Interaction, pp. 483–503. Lawrence Erlbaum Associates (2003)
14. Ramdoyal, R., Cleve, A., Hainaut, J.-L.: Reverse Engineering User Interfaces for Interactive Database Conceptual Analysis. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 332–347. Springer, Heidelberg (2010)
15. Ricca, F., Scanniello, G., Torchiano, M., Reggio, G., Astesiano, E.: On the effectiveness of screen mockups in requirements engineering. In: 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (2010)
16. Rivero, J.M., Grigera, J., Rossi, G., Robles Luna, E., Koch, N.: Towards agile model-driven web engineering. In: Nurcan, S. (ed.) CAiSE Forum 2011. LNBIP, vol. 107, pp. 142–155. Springer, Heidelberg (2012)
17. Robles Luna, E., Grigera, J., Rossi, G.: Bridging Test and Model-Driven Approaches in Web Engineering. In: Gaedke, M., Grossniklaus, M., Díaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 136–150. Springer, Heidelberg (2009)
18. Robles Luna, E., Rossi, G., Garrigós, I.: WebSpec: a visual language for specifying interaction and navigation requirements in web applications. Requir. Eng. 16(4), 297–321 (2011)
19. 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)
20. Ton, H.: A Strategy for Balancing Business Value and Story Size. In: AGILE 2007 Conference (2007)
21. Valderas, P., Pelechano, V., Pastor, Ó.: Introducing Graphic Designers in a Web Development Process. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007. LNCS, vol. 4495, pp. 395–408. Springer, Heidelberg (2007)