Journal of Web Engineering, Vol.9, No.2 (2010) 132-156

INCORPORATING USABILITY REQUIREMENTS IN A TEST/MODEL-DRIVEN WEB ENGINEERING APPROACH

ESTEBAN ROBLES LUNA [2,3], JOSÉ IGNACIO PANACH[1], JULIÁN GRIGERA[2],

GUSTAVO ROSSI [2,3], OSCAR PASTOR[1]

[1]*Centro de Investigación en Métodos de Producción de Software*

Universidad Politécnica de Valencia

Camino de Vera s/n, 46022 Valencia, Spain

{jpanach, opastor}@pros.upv.es

[2]*LIFIA, Facultad de Informática, UNLP, La Plata, Argentina*

{esteban.robles, julian.grigera, gustavo}@lifia.info.unlp.edu.ar

[3]Also at Conicet

Received November 13, 2009
Revised May 26, 2010

The success of Web applications is constrained by two key features: fast evolution and usability. Current Web engineering approaches follow a "unified" development style which tends to be unsuitable for applications that need to evolve fast. Moreover, according to the quality standard ISO 9126-1, usability is a key factor to obtain quality systems. In this paper, we show how to address usability requirements in a test-driven and model-based Web engineering approach. More specifically, we focus on usability requirements with functional implications, which do not only concern the visual appearance, but also the architecture design. Usability requirements are contemplated from the very beginning of each cycle, by creating a set of meaningful tests that drive the development of the application and ensure that no functionality related to usability is altered unintentionally through development cycles. Dealing with those usability requirements in the very early steps of the software development process avoids future hard changes in the system architecture to support them. The approach is illustrated with an example in the context of the OOWS suite.

Key words: Test-Driven Development, Usability, Conceptual Models, Model-Driven Development
Communicated by: M. Gaedke & A. Ginige

1 Introduction

Developing quality Web applications quickly and error free is one of the most challenging problems in the web engineering field. This kind of software always stresses development teams because requirements tend to change fast (the “permanent beta” syndrome) [25]. At the same time, customers require extremely usable applications more than in other kind of software. As a consequence, it is reasonable to use a development process with short cycles and intense participation of stakeholders.

Agile development methodologies, such as Test-Driven Development (TDD) [3, 16], are a perfect match to this development style.

TDD uses short cycles to incrementally add behaviour to the application. Each cycle starts by gathering requirements in the form of Use Cases [19] or User Stories [21] that describe the expected behaviour of the application informally. Next, the developer abstracts concepts and behaviour, and concretizes them in a set of meaningful test cases. Those tests are intended to fail on their first run, showing that the application does not meet the requirements yet. In order to fix this problem, the developer writes the necessary code to pass the tests and runs them again until the whole test suite passes. The process is iterative and continues by adding new requirements. In these cycles, the developer can refactor [14] the code when it is necessary. Studies have shown that TDD reduces the number of problems in the implementation stage [33] and therefore its use is growing fast in industrial settings [26].

However, one of the problems of TDD is its extremely informal nature in which most design decisions remain undocumented. While TDD favours agility, it also hinders evolution in the middle and long term.

One attractive alternative to standard "code-based" TDD is to use a model-driven software development (MDSD) approach, which allows focusing on higher level design models and deriving code automatically from them, at the same time minimizing errors and making the development process faster [15]. However, MDSD Web engineering approaches [23, 6, 13, 17, 35] tend to use a "unified" [20] rather than an agile approach. To make matters worse, both agile and MDSD approaches lack a "natural" way to specify requirements dealing with usability, which as mentioned before is a key aspect in the Web engineering field.

While agile and MDSD-based approaches appear to be confronted frequently, our view is that their positive properties should be put to work together in order to provide more efficient and effective software production methods. This is why in this paper we present a novel development approach which aims to solve the problems discussed above: it is agile, can interplay seamlessly with model-driven approaches and supports specification and testing of usability requirements. Our approach combines the recent work of the authors in two different areas: test-driven development of Web applications [34], and specification and modelling of usability requirements [30].

On the one hand, we propose injecting a test-driven development style into a model-driven development methodology, developing the initial ideas presented in [34]. In this way, we maintain the agility of test-driven development while working at a higher level of abstraction by using models. One contribution of the presented work is to show that Agile and MDSD can be combined to make them become stronger together than separately. The approach begins building interaction and navigation tests derived from presentation mockups (i.e. stub HTML pages) and User Interaction Diagrams (UIDs) [35]; these tests are later run against the application generated by the model-driven development tool to check whether they pass or fail. On the other hand, we derive usability tests, i.e. those that capture the properties needed to build a usable system. These tests are used in the same way as "conventional" functional tests in the TDD cycle, thus serving as a way to check how development proceeds by formalizing one of the typical customers concerns. Another value of the presented work is to demonstrate that usability requirements can be properly dealt with in such an advanced software production process. The whole approach is complemented with a set of tools to simplify the stakeholders' tasks.

To develop this idea, our work focuses on functional usability requirements, called in the literature Functional Usability Features (FUF) [22]. Historically, usability has been considered as a non-functional requirement [7]. However, many authors have discovered several usability properties strongly related to functionality [2, 12]. FUFs are usability requirements related to functionality and therefore related to system architecture. Each FUF is divided into different subtypes called usability mechanisms. Several authors propose including these mechanisms from the very early steps of the software development process in order to avoid changes in the architecture once they have been designed [2, 12]. Following the proposal to deal with usability in the early steps, we have used a set of guidelines defined by Juristo [22] to capture functional usability requirements for each usability mechanism. These templates contain a set of questions that the analyst must use to capture usability

requirements by means of interviews with the client. From these templates, we have extracted the usability properties that must be taken into account when the analyst develops the system.
In a brief summary, the contributions of the paper are the following:

- We show how to introduce usability requirements in an agile model-driven Web engineering approach.
- We illustrate the detection of some relevant properties to build usable systems. These properties have been extracted from templates to capture usability requirements defined in the literature.
- We show how to translate those properties into a set of meaningful tests that drive the development process.

The structure of the paper is the following: In Section 2 we discuss some related work in this area, covering Web development approaches and specification of usability requirements. In Section 3 we present the background of our approach. In Section 4 we show the approach in a step by step way, showing with small examples how we intermix test and model-driven development with a strong bias to usability checking. In Section 5 we present a lab case. Finally, in Section 6 we conclude and present some further work we are pursuing.

2 Related Work

Our proposal brings model-driven and agile approaches together, in an effort to improve Web development. Classical model-driven Web engineering methods like WebML [6], UWE [23], OOHDM [35], OOWS [13] or OOH [17] usually favour a cascade style development. We superimpose a specific agile approach, Test-Driven Development, where tests are developed before the code (in this case the model) in order to guide the system development. In this sense, some authors like Bryc [4] have proposed generating these tests automatically, while in other works tests are constructed manually [26]. Both techniques are valid for our proposal.

We state, like Bass [2] and Folmer [12], that usability must be included from the very early steps in the software development process (TDD in our proposal). In other words, usability must be considered from the requirements capture step. Several authors, like Juristo [22], have dealt with usability as a requirement. Juristo has defined a set of Functional Usability Features that are related to system architecture. The requirements of these features are captured by means of guidelines. These guidelines include questions that the analyst must ask to end-users in order to adapt the features to users' requirements. Lauesen [24] also includes usability in the requirements capture, discussing six different styles of usability specification and showing how to combine them in a complex real-life case to meet the goals. The styles specify the usability properties more or less directly. The list of styles is: performance style; defect style; process style; subjective style; design style; guideline style. The best choice in practice is often a combination of the styles, so that some usability requirements use one style and others use a different one. Finally, it is important to mention the work of Cysneiros [8], who has defined a catalogue to guide the analyst through alternatives for achieving usability. The approach is based on the use of the i* [42] framework, having usability modelled as a special type of goal. Cysneiros states that a catalogue can be built to guide the requirements capture. This notation provides a total view of requirements and the relationships among them, as well as the relationship between usability and functional requirements inclusively. The main disadvantage of this proposal is the i* notation which is ambiguous, is far from natural language, and it may present contradictions [11].The difference between our proposal and the aforementioned works is the context of use. We deal with usability requirements in a TDD process using a model-driven Web engineering approach, while the mentioned authors deal with usability requirements in a traditional software development process.

The concept of pattern is one of the most widely used concepts to include usability in the first steps of the software development process. Many authors have worked on the definition of usability patterns, for instance Tidwell [38]. The patterns described by Tidwell represent not only usability, but also interaction. The notation used to represent the patterns is graphical because Tidwell wants the user

to participate in the design of the architecture. Following the same trend as Tidwell, Perzel describes a set of patterns that are oriented to web environments [32]. Perzel distinguishes between patterns for web applications (users must introduce data) and patterns for web sites (users only navigate and visualize information).

One work that aims to bring usability patterns closer to the end user is carried out by Welie [40]. The patterns of Tidwell and Perzel differ from the patterns of Welie in that Welie distinguishes between the user perspective and the design perspective. The main reason for this sorting into groups is that, from the user perspective, it is important to state *how* and *why* the usability is improved; while from the design perspective, patterns only solve designer problems.

The patterns proposed by all these authors include a short description about the implications of including the patterns in the architecture. However, this description is too short. The patterns should have a guideline to explain in detail how to include the patterns in the system. In our proposal, that inclusion is hidden for the analyst, because it is performed by automatic transformation of the MDSD process.

Others authors like Nielsen [29], have been working recently on including usability in agile software development methods. Nielsen states that fast and cheap usability methods are the best way to increase user experience quality, because developers can use them frequently throughout the development process. This work is very close to our proposal, but it is not focused on a TDD approach. Again, the originality provided by our presented work is centered around its integration of TDD and MDSD, together with the incorporation of usability requirements in this approach.

Regarding automated testing within model-driven software development processes, it is important to mention the work of Dihn-Trong et al. [9] who apply validation techniques directly to UML models [39]. The authors create Variable Assignment Graphs (VAGs) to automatically generate test input, considering also the model's constraints. Nevertheless, generating VAGs requires the models to be already created, so it is not possible to guide the development through generated tests. Also, we state that users must participate in the test definition, but Dinh-Trong proposes testing the system by means of design models, where users cannot take part for ignorance.

In a recent work [34] we have illustrated a first attempt to apply our TDD-based methodology on a MDSD Web engineering approach, but usability was not considered. In the same way, Zhang [45] has presented an approach in which he applies Extreme Programming practices into a process, in a methodology called test-driven modelling (TMD). Tests are created in terms of message sends (represented as Message Sequence Charts) to a black box system, and then models are created to pass these tests. The overall approach is similar to the one here presented, but it does not consider navigation or presentation (hence, neither usability) early in the process.

Back on the agile track, Agile Model Driven Development (AMDD) [1] proposes a MDSD-like development process, but creating models that are “just barely good enough” to fulfil a small set of requirements. Our approach takes the same philosophy in that matter, but AMDD differs from it since it is not purely model-based, but it also has a latter coding stage in which TDD is applied. Other authors, such as Wieczorek [44], have proposed testing the system in the code generated from a Conceptual Model, as we propose. This author proposes black-box testing that uses structural and behavioural models described in UML to automatically generate test cases. After automatically generating part of the code from the Conceptual Model, developers are starting to create unit tests for the functions that they are going to implement. Changes derived from testing are applied directly to the code. This fact differs from our proposal, where changes are directly applied to the Conceptual Model and the code is automatically generated, making the software development process more efficient.

3 Bridging Usability requirements with TDD

We want to emphasize that our approach [34] puts together the advantages of both agile and model-driven approaches, and it is our strong belief that this is the path to be followed by modern software production approaches. Incorporating usability requirements in that domain is a concrete way to improve the quality of the associated method, as usability is a recognized quality software criteria. To achieve this goal, we deal with presentation mockups and requirements models early in the

requirements elicitation stage, integrating usability requirements in the incremental development process with a TDD style. By using a model-driven development approach we raise the level of abstraction; as a consequence, the quality of the generated software is significantly improved like in most MDWE approaches [15]. Instead of following a cascade style of development, we use an iterative and incremental style following short cycles constantly involving stakeholders. Usability requirements are taken into account from the earliest stages, and considered "first class" requirements for test generation; therefore they participate in the development cycle, just as regular requirements. We next describe our approach in detail.

3.1 The approach in a nutshell

The development cycle is divided into cycles or sprints (Figure 1). At the beginning of the sprint, the development team has only a set of short informal specifications (descriptions) of what they have to do. These specifications have been defined by means of interviews with the user. Developers start working by picking one of them at a time. A small cycle starts by capturing a more detailed analysis using informal requirement artefacts (Step 1). A variety of artefacts can be used depending on the type of requirement we are capturing:

- For requirements involving interactions, we use UIDs that serve as a partial specification of the application's navigation. Mockups are used for User Interface (UI) aspects, and Use Cases (UC) or User Stories (US) for business or domain aspects.
- For usability requirements, we use a set of usability properties derived from usability requirements guidelines defined in the literature [22]. These usability properties are represented by: UIDs for navigational concerns and mockups for UI aspects. If functionality slightly changes, then UC/US must be used too.

The artefacts we use to capture requirements are described in natural language, lacking a clear/formal definition. Therefore, developers transform these requirements into tests, to get a more "formal" specification (Step 2). As in a TDD, tests are heavily used both to drive the development process and to check that existing functionality is not altered during the development process. This has probed [28] to reduce development time because unintentional errors are captured during the development cycle instead of leaving their discovery to the quality assurance (QA) or testing phase.

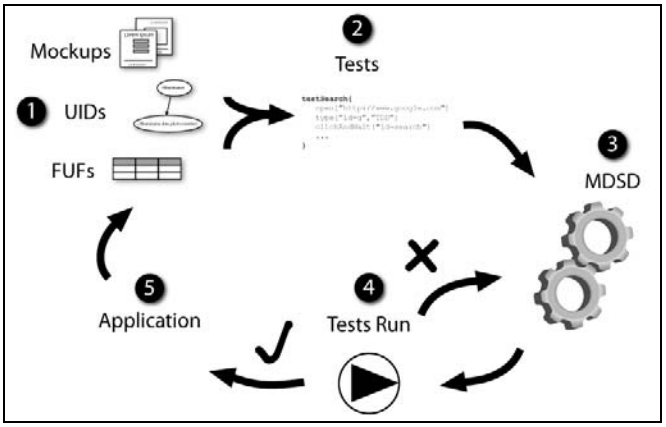


Figure 1 A Schema of our method

Before a requirement is implemented, corresponding tests must be run to check whether or not the application fulfils the requirement. The failing tests show which requirements are not yet supported by the system under development. If, at this stage, the application passes all the tests, then either they do not express the new requirement properly, or the new requirement is not new because the application already supported it. In the first case, we should give more detail to tests going back to step 2 and in the second one, we should dismiss the requirement and return to step 1.

Once the requirement has been specified in a test suite, the development phase can begin. By using a MDSD approach, the developer creates or extends the existing models generating an enhanced version of the application (Step 3). All the development effort is concentrated on building/extending the model. The code generation is performed automatically by means of transformation that takes as input the models.

In order to check that the requirement has been successfully implemented and no previous functionality is corrupted, the developer runs the whole suite of tests to check both things (Step 4). If one or more tests fail, he should go back to step 3, do some rework in the models, generate the code again, and retry step 4 until all tests pass.

Finally, we get a new application with one requirement added (Step 5). The cycle continues by picking a new requirement (Step 1) and following steps 2 to 5 until we run out of requirements for the sprint.

3.2 An overview of involved artefacts

Throughout the requirements elicitation activity, we combine different artefacts to achieve fluency in the communication between stakeholders, and accuracy in the specification for the development team. As we just stated in 3.1, UIDs, HTML mockups, FUFs and interaction tests help in both aspects. The first two are useful in terms of communication at early stages of requirements definition: UIDs provide a precise and somewhat intuitive way to specify navigation and interaction, while mockups reveal the presentation, making it concrete for customers. Usability needs are also detected at this early stage, following standardized guidelines by applying Functional Usability Features. Whenever possible, these requirements must be stated early in the process, since they might have an influence in the application's design (particularly in navigation and interface/interaction issues, but also in functionality). Finally, interaction tests come to play right before development, when a thorough specification that considers all possible ways of interaction stated in the UID diagrams is required. Additionally, they are specified against the same mockups obtained in the requirements gathering activity, to make sure the same interaction agrees with the stakeholders.

Along the following subsections, we explain each artefact with more detail, illustrating them with examples in the context of a simple, conventional e-commerce application that is useful to introduce the basic ideas. In section 5, we will introduce in more detail a concrete example related to a library management system.

3.2.1 Mockups

HTML mockups are simple, static web pages that act as sketches of the application. They are intended to be developed quickly to reflect the customer wishes in terms of presentation in a much more substantial way than requirements expressed in written language alone. Mockups show no difference from regular HTML pages, in fact their only characteristic feature is the way of building them and their use. However, they can eventually become useful in the final stages of the development, where the same mockups can become the definitive look and feel of the application.

In a simple E-commerce application, suppose that the customer explains that the checkout process must ask for the credit card information, and let the user revisit the list of products involved in the purchase. Figure 2 shows two mockups, Figure 2.a shows a product's detail page, from which the user can navigate to the checkout page shown in Figure 2.b.

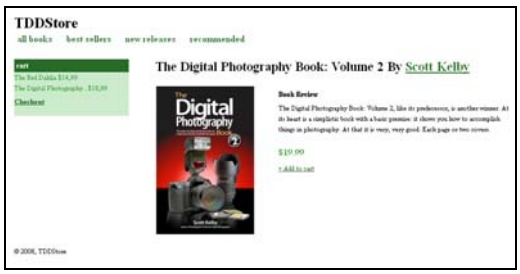


Figure 2a Product Detail Mockup

Figure 2b Checkout Mockup

3.2.2 User Interaction Diagrams

Similar to use cases, a UID describes the exchange of information between the user and the system and particularly the set of interactions occurring to complete a functional requirement. UIDs enrich use cases with a simple graphical notation to describe partial navigation paths. UIDs are simple state machines where each interaction step (i.e. each point in which the user is presented with some information and either indicates his choice or enters some value) is represented as an ellipse, and transitions between interaction points as arcs. 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. Following the example from 3.2.1, Figure 3 shows a simple UID expressing the operation of checkout, from the list of products through the confirmation.

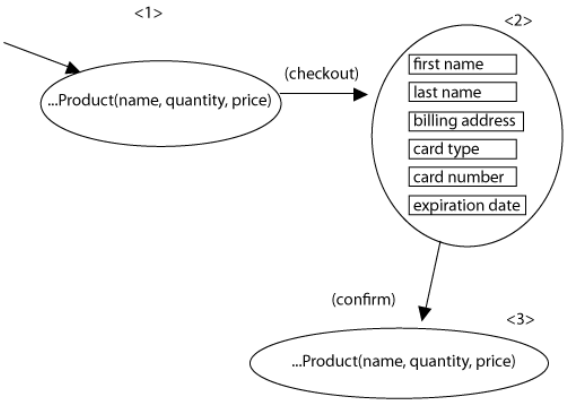


Figure 3 Checkout's UID

We have extended the UID notation to allow automatic generation of interaction tests as described below.

3.2.3 Interaction tests

An interaction test is a test that opens a Browser and executes a set of actions directly on it, in the same way a user would do it. Also it allows making assertions on HTML elements based on XPath [41] or HTML IDs There are several tools that could be employed to write such a test: Selenium [36], Watir [43], TestNG [37]. The main advantage of this kind of tests is that they execute directly in the browser making them independent of the development method or tool used to generate the application.

Like in "conventional" TDD, we write the tests before the development begins. Using Selenium, we can speed up the code writing by profiting from the Selenium recorder to record the set

of actions we perform over the mockup. Later, we can translate the tests to Java and add the necessary assertions to ensure the application's expected behaviour.

These tests are also used after the requirement has been developed to check that the new requirement has been correctly implemented and previous functionality has not been altered.

3.2.4 FUF

Usability is a very wide concept. Human-Computer Interaction literature provides three types of recommendations to improve the usability of a software system [22].

1. Usability recommendations with impact on the user interface (UI). These recommendations refer to presentation issues with slight modifications of the UI design (e.g. buttons, pull-down menus, colours, fonts, layout).
2. Usability recommendations with impact on the development process. These can only be taken into account by modifying the whole development process. For example, those that intend to reduce the user cognitive load require involving the user in the software development.
3. Usability recommendations with impact on the architectural design. These involve building certain functionalities into the software to improve user-system interaction. These set of usability recommendations are referred to as Functional Usability Features (FUFs). FUFs are defined as recommendations to improve the system usability that have an impact on the architectural design. Examples of these FUFs are providing cancel, undo and feedback facilities.

We have focused our proposal on FUFs because a big amount of rework is needed to include these features in a software system unless they are considered from the first stages of the software development process [2, 12]. Therefore, the inclusion of FUFs must be done from the requirements capture step.

Different HCI authors [40, 38, 18] identify different varieties of these usability features. These subtypes are called usability mechanisms. In other words, each FUF has a main goal that can be specialized into more detailed goals called usability mechanisms.

3.3 OOWS: A Model-Driven Web Engineering Method

As said before, we favor the use of a MDSD style. Though the overall approach is independent of the specific MDSD methodology, we will illustrate the paper with OOWS. OOWS (Object-Oriented Web Solutions) [13] is a model-driven web engineering method that provides methodological support for web application development. OOWS is the extension of an object-oriented software production method called OO-Method [31], as Figure 4 illustrates. OOWS introduces the diagrams that are needed to capture web-based applications requirements, enriching the expressiveness of OO-Method. OO-Method is an Object Oriented (OO) software production method that provides model-based code generation capabilities and integrates formal specification techniques with conventional OO modelling notations. OO-Method is MDA compliant [27], so following the analogy with MDA, OO-Method provides a PIM (Platform-Independent Model) where the static and dynamic aspects of a system are captured by means of three complementary views, which are defined by the following models:

- **Structural Model** that defines the system structure and relationships between classes by means of a *Class Diagram.*
- **Dynamic Model** that describes the valid object-life sequences for each class of the system using *State Transition Diagrams.*
- **Functional Model** that captures the semantics of state changes to define service effects using a textual formal specification.

As Figure 4 shows, OOWS introduces two models in the development process:

- **Navigational Model**: This model describes the navigation allowed for each type of user by means of a Navigational Map. This map is depicted by means of a directed graph whose nodes represent navigational contexts and their arcs represent navigational links that define the valid navigational paths over the system. Navigational contexts are made up of a set of **Abstract Information Units** (AIU), which represent the requirement of retrieving a chunk of related information. AIUs are made up of **navigational classes**, which represent views over the classes defined in the Structural Model. These views are represented graphically as UML classes that are stereotyped with the *«view»* keyword and that contain the set of attributes and operations that will be available to the user. Basically, an AIU represents -at a conceptual level- a web page of the corresponding Web Application.
- **Presentation Model**: The purpose of this model is to specify the visual properties of the information to be shown. To achieve this goal, a set of presentation patterns are proposed to be applied over conceptual primitives. Some properties that can be defined with this kind of patterns are information arrangement (register, tabular, master-detail, etc), order (ascendant/descendent) or pagination cardinality.

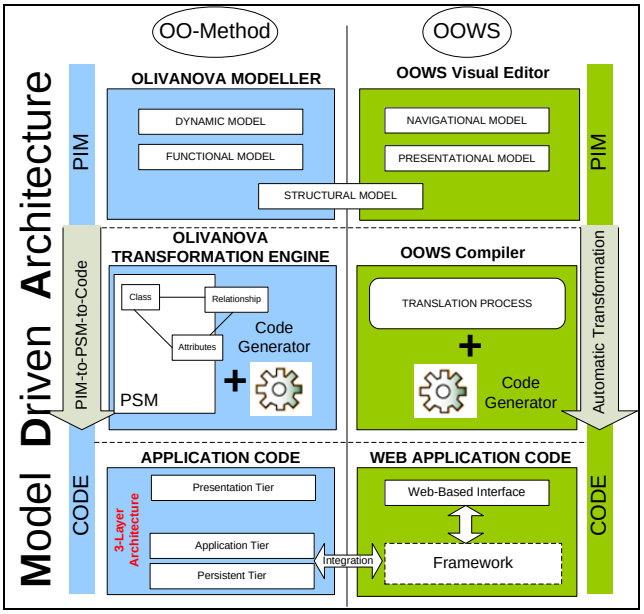


Figure 4 OO-Method and OOWS MDA Development Process

Both models are complemented by OO-Method models that represents functional and persistence layers. OOWS generates the code corresponding to the user interaction layer, and OLIVANOVA [5], the industrial OO-Method implementation, generates the business logic and the persistence layers.

4 The approach in action

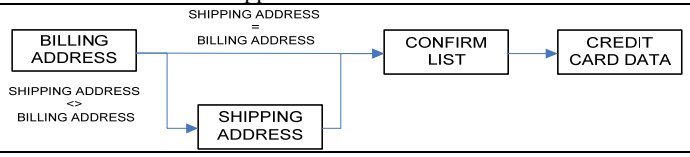
To illustrate the approach we will show concrete examples of usability requirements following the same basic notions commented in section 3.2. Functional requirements have already been dealt with in

[34], so for the sake of conciseness, we will stress how to deal with usability requirements and show only a couple of UIDs.

Assuming we have already developed the concepts of *product*, *category* and *list of products* in the E-commerce application, we are about to start a new sprint that concentrates in the *shopping cart*. We will show the approach in the context of improving the usability of the existing checkout process which is performed in a single web page. Specifically, we want to include a wizard to carry out the checkout process. To do so, we are going to use a FUF called Wizard. This FUF has a usability mechanism called Step by Step which has the goal of helping the user in complex tasks. We will go through the following steps to develop this requirement:

1. We extract the usability properties from the requirements guideline of the usability mechanism called Step by Step (Table 1). Those properties specify the service that will be executed at the end of the wizard; how we have to split the navigation concern; the description for each step; how the information will be displayed in each step, and whether or not each step will inform about the number of remaining steps. We need to refactor the current mockups to show what we expect from a UI perspective.

Table 1. Usability properties for Step by Step

Step by Step	
Property	**Value specified by the analyst in the checkout example**
Service selection	This mechanism will be applied to the *checkout action*
Steps division	
Step description	Each step must contain a short description
Visual aspect	The user has specified the widgets to fill in each step
Remaining steps	The system must inform about the number of remaining steps

2. We capture the navigation between the different steps of the checkout process in a UID that will serve as a partial navigation model (see Figure 5.a), allowing the developer to implement the navigation aspect of the requirement. Then, we rework on the mockup of the checkout process by splitting it into several steps (Figure 5.b shows some resulting mockups for these steps). We add the necessary widgets as described in the table 1 to show the remaining steps and their descriptions (on each node).

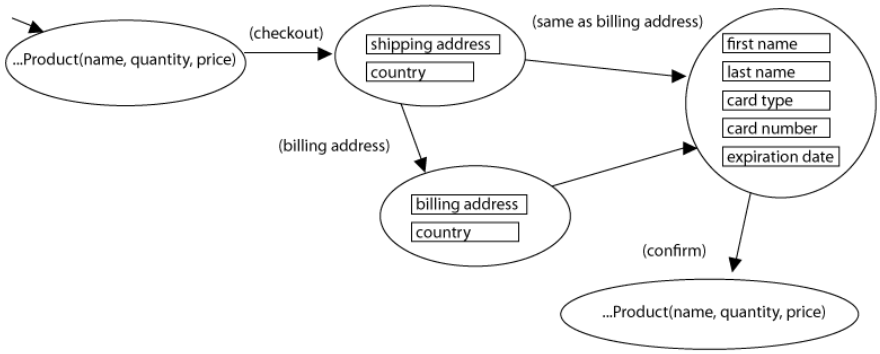


Figure 5a UID for checkout steps.

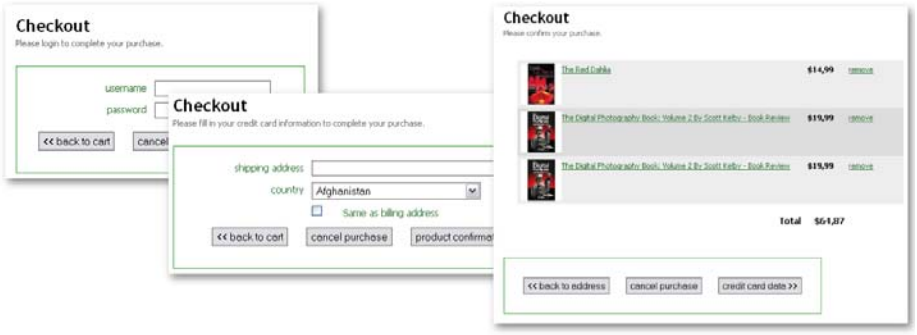


Figure 5b Sample mockups for checkout steps.

3. We refactor the existing checkout test so that it specifies the new process. We have to add the necessary asserts to validate the description, remaining steps, etc. Then, we run the test to check whether it is a new requirement and the application does not support it yet. We next show a test in Selenium Java notation:

```
public class CheckoutTestCase extends SeleneseTestCase {
    public void testSuccessfulCheckout() throws Exception {
(01) sel.open("file:///dev/bookstore/Mockups/books-list.html");
(02) sel.clickAndWait(
         "/ul[@id='products']/li[1]/div[1]/div[@id='prod-info']/a");
(03) sel.assertLocation("/cart*");
(04) assertEquals(
         "The Digital…",
         sel.getText("/ul[@id='selected-products']/li[1]/span[1]"));
(05) sel.clickAndWait("checkout");
(06) sel.assertLocation("/checkoutStepShippingAddress");
(07) assertEquals("3", sel.getText("//div[@id='remaining']"));
(08) assertEquals(
          "Shipping information",
          sel.getText("//div[@id='stepDescription']"));
(09) sel.type("shipping-address", "Calle 58");
(10) sel.select("country", "label=Argentina");
(11) sel.clickAndWait ("//input[@value='billing information>>']");
(12) sel.assertLocation("/checkoutStepBillingAddress");
(13) assertEquals("2", sel.getText("//div[@id='remaining']"));
(14) assertEquals(
          "Billing information",
          sel.getText("//div[@id='stepDescription']"));
(15) sel.type("billing-address", "Calle 48”);
(16) sel.select("country", "label=Argentina");
(17) sel.clickAndWait ("//input[@value='product confirmation>>']");
(18) sel.assertLocation("/checkoutStepProductConfirmation");
```

```
(19) assertEquals("1", sel.getText("//div[@id='remaining']"));
(20) assertEquals(
          "Product confirmation",
          sel.getText("//div[@id='stepDescription']"));
(21) assertEquals(
         "The Digital…",
         sel.getText("/ul[@id='selected-products']/li[1]/span[1]"));
(22) sel.clickAndWait ("//input[@value='credit card data >>']");
(23) sel.assertLocation("/checkoutStepCreditCardData");
(24) assertEquals("0", sel.getText("//div[@id='remaining']"));
(25) assertEquals(
          "Credit card information",
          sel.getText("//div[@id='stepDescription']"));
(26) sel.type("first-na","Esteban");       sel.type("last-na", "Robles");
(27) sel.type("card-number", "4246234673479");
(28) sel.select("exp-year", "label=2011");
(29) sel.select("exp-month", "label=Apr");
(30) sel.clickAndWait ("//input[@value='confirmation >>']");
(31) sel.assertLocation("/checkoutSucceed");
(32) assertEquals(
          "Checkout succeded”,
          sel.getText("/div[@id='message"));
     }
}
```

The test opens the book list page (1) and adds an item to the shopping cart (2). Then we assert that the book has been added and proceed to the checkout process (3-5). Shipping information (6-11) and billing information (12-17) are filled and confirmed. Products are confirmed by asserting that product's name (18-22). Credit card data is filled (23-30) and then we confirm the process has succeeded by looking at the text displayed in a *div* element (31-32).

4. Since we are modelling the application with OOWS [13] we have to extend the navigation, domain and UI models to fulfil this new requirement. The Conceptual Model Compiler associated to OOWS is the responsible of creating the web application corresponding to the extended models. The strategy followed by it is out of the scope of this paper, but the reader will find the relevant details in [13].
5. We check that the application obtained in step 4 satisfies the requirements by running the whole test suite. If one test fails then we have to go back to step 4.
6. We get a new version of the application by integrating new changes with the current version of the model.

4.1 Handling usability requirements

As we have previously seen, usability requirements with functional implications have already been catalogued in the literature [22] as Functional Usability Features (FUF). These FUFs are derived from usability heuristics, rules and principles. In other words, FUF are functional requirements that improve particular usability attributes. Moreover, FUFs are divided into different specialised subtypes called

usability mechanisms. The definition of these mechanisms includes a set of guidelines to lead the analyst in the usability requirements capture. The guidelines are composed of questions that the analyst must ask to the user in order to extract usability requirements. We have used those guidelines to identify the usability properties that must be considered in the early stages of the software development process. Properties are the different configuration possibilities that a usability mechanism has to adapt itself to usability requirements. More details about our proposal to extract usability properties from usability mechanisms guidelines can be found in [30].

In Figure 6 we show a sketch to explain how we have extracted usability properties from FUFs. Each FUF is divided into several usability mechanisms. These mechanisms include a guideline to capture usability requirements. From those guidelines, we have extracted a set of properties. Grey boxes in Figure 6 represent existing elements in the literature, while white boxes represent a new contribution of our work.

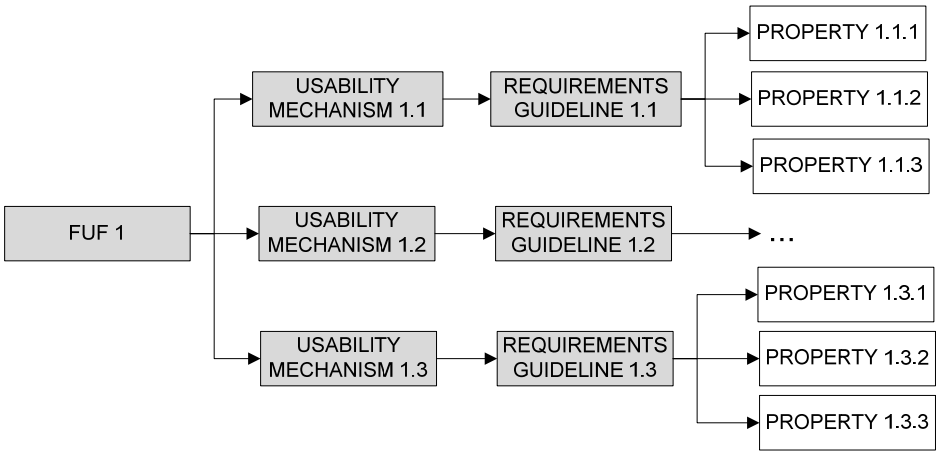


Figure 6 Division of FUF into properties

In this section, we show usability properties extracted from the definition of some usability mechanisms. Specifically, we focus on mechanisms related to Web applications environments. For each mechanism, we have specified its motivation according to existing works [22], the properties derived from it, and finally an overview of how those properties must be mapped to a concrete test suite. The test suite and the different scenarios are going to be explained in natural language because of the nature of FUFs: they are general and not concrete for a specific application. Later in the lab case presented in section 5 we are going to show how those scenarios are mapped in concrete software artefacts.

4.1.1 Favourites

The motivation of this mechanism is to let the users make a record of their points of interest, so that they can easily go back to them later. This mechanism allows the user to move freely through a way that is not directly supported by the structure of the system. The context of use of this mechanism is interfaces that the user visits frequently. Properties derived from the requirement guideline of Favourites are the following:

- Favourites' location: This property is used to specify where the list of favourites will be shown in the interface. For example, the list of favourites can be included in the main menu, in the main interface, in a specific window, etc.
- Num of items: This property specifies the maximum number of items listed in the favourites' area. Analyst must adapt this property according to the interface size and the user's requirements.

Test generation

The test suite for Favourites must include the following scenarios:

- Add favourite: Open the application. Navigate to a specific item that wants to be categorized as favourite. Add to favourite list. Check that the item is added to the favourite list.
- Navigate to favourite: Open the application. Identify as a user that already has favourites. Click on a favourite. Check that navigation has occurred to the specific item.
- Validate favourites' location: Open the application. Check the location of the favourites' area using an XPath expression.
- Validate number of items: Open the application. Add n + 1 (n is maximum) items as favourites. Check that favourite's area only contains n items.

4.1.2 Progress Feedback

The motivation of this usability mechanism is providing users with information related to the evolution of the requested services. The concept of service is defined as a processing unit that modifies the local state of an object according to [31]. The context of use for this mechanism is when a process interrupts the user interface for longer than two seconds. In that case, the system should show an animated indicator of how much progress has been made. Properties derived from the requirement guideline of Progress Feedback are the following:

- Service selection: This property is used to select which services will show the progress of their execution. Analyst must select the services that usually will spend more than two seconds in the execution.
- Visualization options: Analyst uses this property to decide how the progress will be shown to the user. This progress can be shown by means of a progress bar or by a list of completed services. Moreover, both types of visualization have several options. For example, the progress bar can be shown from right to left, from left to right, including the remaining time, including the remaining percentage, etc.

Test generation

The test suite for Progress Feedback must include the following scenarios:

- Check progress existence: Open the application. Execute a service that requires progress. Check the presence of the progress bar. Wait for the result to be loaded. Check that the result is properly loaded.
- Check services (optional in case those services must be shown): Open the application. Execute a service that requires progress. Check the presence of the progress bar and the list of completed services is shown. Check that the result is properly loaded.

4.1.3 Abort Operation

The motivation of this usability mechanism is to provide a way of cancelling the execution quickly. The functionality of Abort Operation consists in interrupting the processing and going back to the previous state. The context of use is when a process interrupts the user interface for longer than two seconds. In that context, the system should provide a mechanism to cancel the execution. Properties derived from requirements guideline of Abort Operation are the following:

- Service selection: This property is used to select which services can interrupt their execution when the user considers. The analyst must select those services whose execution will spend more than two seconds or services that can block the system.
- Visualization options: This property is used to specify how the abort option will be shown to the user. For example, this functionality can be accessed by means of a button in the main menu, a button in an emergent window together with the progress bar, a short cut, etc.

Test generation

The test suite for Abort must include the following scenarios:

- Check abort cancelled: Open the application. Execute a service that requires cancel. Click the cancel button. Wait for the page to be loaded. Check that a message saying that the operation has been cancelled is shown.
- Check abort bypassed: Open the application. Execute a service that requires cancel. Wait for the page to be loaded. Check that the service has been executed.

4.1.4 Warning

The motivation of this usability mechanism is to ask for user confirmation in case the service requested has irreversible consequences. The context of use is when a service that has serious consequences has been required by the user. The properties derived from the requirements guideline of Warning are:

- Service selection: The analyst must choose which services have irreversible consequences depending on the business logic.
- Condition: This property is used to specify when the warning message must be shown. The analyst must define this condition using stored information and data written in input fields.
- Visualization options: This property is used to specify how the warning message will be shown to the user. For example, the text format, whether the message will appear in an emergent window or not, etc.

Test generation

The test suite for Warning must include the following scenarios:

- Check warning bypassed: Open the application. Fill the necessary data to make the warning happened. Execute the service. Check the presence of the warning. Accept the warning and check that the service has been executed.
- Check warning cancelled: Open the application. Fill the necessary data to make the warning happened. Execute the service. Check the presence of the warning. Cancel the warning and check that the service has NOT been executed.

4.1.5 Structured Text Entry

The motivation of this usability mechanism is to guide the user when the system can only accept inputs from the user in an exact format. The context of use is widgets that require a mask to guarantee the correct format in the data entry. The properties derived from the requirements guideline of Structured Text Entry are:

- Widget selection: This property is used to choose the services that need a mask. The analyst must decide which input fields will have a mask according to the business logic.

- Regular expression: This property is used to specify the format that the widget requires. The format is specified by means of a logic expression.

Test generation

The test suite for Structured Text Entry must include the following scenarios:

- Check widget set invalid: Open the application. Navigate to the selected node. For each widget: Add invalid input to it. Try to execute the service. Check the presence of the error message.
- Check widget set valid: Open the application. Navigate to the selected node. For each widget: Add valid input to it. Try to execute the service. Check that the error is not present for the specific widget.

5 A Lab Case

As a lab case example we have selected a Web application of a library. This system is used to perform distantly the most frequent actions in a library by means of internet. More specifically, we focused our work on three functionalities: Opening the main window (home); looking for a specific book; renewing the loan. Figure 7 represents the OOWS model for the contexts of those three functionalities. Following, we are going to explain the usability properties that the system must support and the tests that we have defined to guarantee those properties. In order to simplify how the examples are shown, we are not going to show each test scenario; only the most significant ones. We have divided the explanation into usability mechanisms, and for each usability mechanism we show a simplified version of the final user interface where the value of the usability properties can be seen applied in the system.

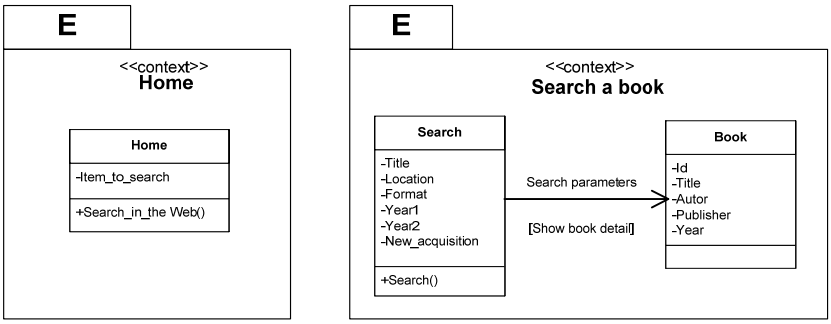


Figure 7a OOWS model for home and search a book

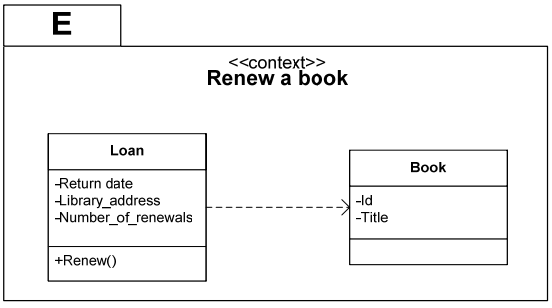


Figure 7b OOWS model for renew a book

In some cases, applying the usability mechanisms lead to changes not only in UI but in all three OOWS models, for example forcing us to change or grow the application's navigation structure. In such cases, the generated tests are again used as guide for the development through the changes in the navigation and domain models.

5.1 Favourites

According to the user requirements, the Web application should include in the main window a list of the most used interfaces. Therefore, the user can visit those interfaces directly from the main window, doing the user's work more efficient. The values for the usability properties of Favourites are shown in Table 2

Table 2. Usability properties for Favourites

Favourites	
Property	**Value specified by the analyst in the library case study**
Favourites' location	In the main window, in the right bottom
Num of items	Three items

Figure 8 shows a mockup of the main window where the list of favourites appears in the right bottom (*pages most visited*).

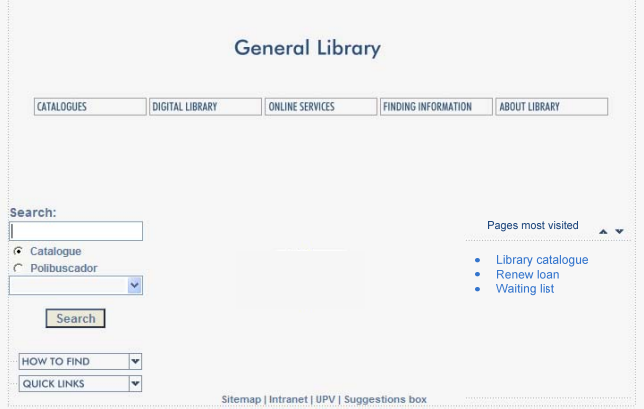


Figure 8 Mockup of the main window with favourites list

Test generation

Add favourite

```
Open("http://library.upv.es") //Open the application
ClickAndWaitPageToLoad("id=recomm1") //Open the first recommendation
AssertElementPresent("id=favourite0") //Assert that the recommendation has been
     added
```

Navigate to favourite

```
Open("http://library.upv.es") //Open the application
Type("id=username", "Pablo") //Authenticate with an existing user
Type("id=password", "apasd")
```

```
ClickAndWaitPageToLoad ("id= login")
ClickAndWaitPageToLoad ("id= favourite0") //Navigate to a favourite
AssertLocation("book.asp") //Assert that navigation has occurred
```

Validate Num or items

```
Open("http://library.upv.es") //Open the application
For (I in 1..4) {
 ClickAndWaitPageToLoad("id=recomm1") //Open the first recommendation
 ClickAndWaitPageToLoad ("id=home") //Go back to home
}
//Assert that 3 elements are shown
For (I in 1..3) {
 AssertElementPresent("id=favourite" + I)
}
AssertElementNotPresent("id=favourite4") //Assert that the 4th element is not shown
```

5.2 Progress Feedback

The process of looking for a specific book can take several seconds. According to usability requirements, the system should inform the user that the search service is in progress and how much time the user must wait until the search finishes. The values for the usability properties of Progress feedback are shown in Table 3.

Table 3. Usability properties for Progress Feedback

Progress Feedback	
Property	**Value specified by the analyst in the library case study**
Service selection	The search service
Visualization options	The system will show a progress bar in an emergent window where the user can see the percentage of the task that has been done. The progress will be drawn from left to right

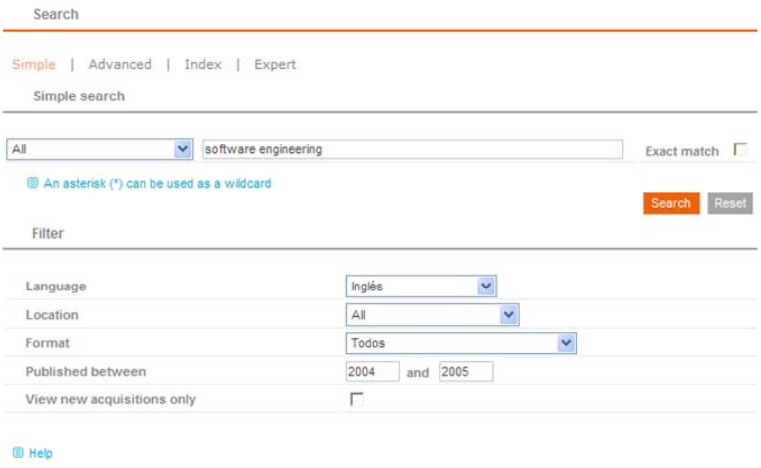


Figure 9 Mockup of the window for searching a book

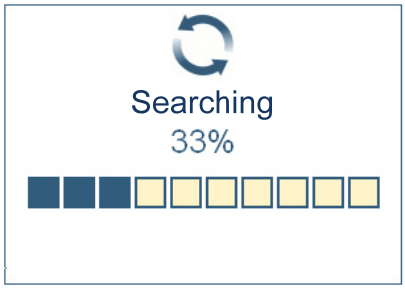


Figure 10 Mockup of a progress bar

Figure 9 shows a mockup where the user can look for a book. In that example, the user wants to look for books of software engineering. When the user presses the search button, a progress bar like Figure 10 will appear. This progress bar shows the percentage of finished task.

Test generation

Check progress existence

```
Open("http://library.upv.es") //Open the application
ClickAndWaitPageToLoad ("id=search") //Go to search
Type("id=searchField", "Development approaches");
ClickAndWaitPageToLoad ("id=doSearch") //Search
AssertElementPresent("id=pBar") //Check that progress bar is shown
WaitPageToLoad(30000)
AssertLocation("searchResults") //Check that search has occurred
AssertTextPresent("Development approaches")
```

5.3 Abort Operation

Once the search has been triggered, the user should be able to cancel this search. Sometimes the search may be too long or the user may have made a mistake triggering the search service. Therefore, the Abort Operation is a requirement for the library Web application. The values for the usability properties of Abort Operation are shown in Table 4.

Table 4. Usability properties for Abort Operation

Abort Operation	
Property	**Value specified by the analyst in the library case study**
Service selection	The search service
Visualization options	The cancel button will be shown in the emergent window together with the progress bar

Figure 11 shows a mockup for the Abort Operation. This usability mechanism has been implemented by means of a cancel button added to the progress bar showed in Figure 10. The user can abort the search service pressing the cancel button.

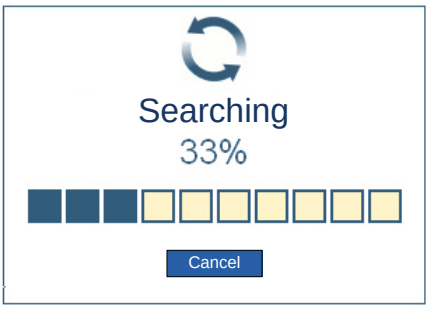


Figure 11 Mockup of a progress bar with abort operation

Test generation

Check abort cancelled

```
Open("http://library.upv.es") //Open the application
ClickAndWaitPageToLoad ("id=search") //Go to search
Type("id=searchField", "Development approaches");
ClickAndWaitPageToLoad ("id=doSearch") //Search
AssertElementPresent("id=pBar") //Check that progress bar is shown
ClickAndWaitPageToLoad ("id=cancel") //Search
AssertLocation("search") //Check that search has been cancelled
```

5.4 Warning

The Web application allows users to renew the loan of a book if there is none in waiting list to get that book. The renewal service let the user have the book one week more and this service can be executed only three times by loan. The renewal service cannot be undone; therefore the execution of the service has irreversible consequences. According to usability requirements, the system must warn the user about the consequences of the renewal service before its execution. The values for the usability properties of Warning are shown in Table 5.

Table 5. Usability properties for Warning

Warning	
Property	**Value specified by the analyst in the library case study**
Service selection	The renewal service
Condition	The warning message must be shown each time the user triggers the renewal service. Therefore the condition is "true" for every case.
Visualization options	The warning message will be shown in an emergent window, with arial font, size 10 and black colour.

Figure 12 shows a mockup to show a warning message when the user triggers the renewal service. Moreover informing the user about the consequences of the execution, the message asks for confirmation.

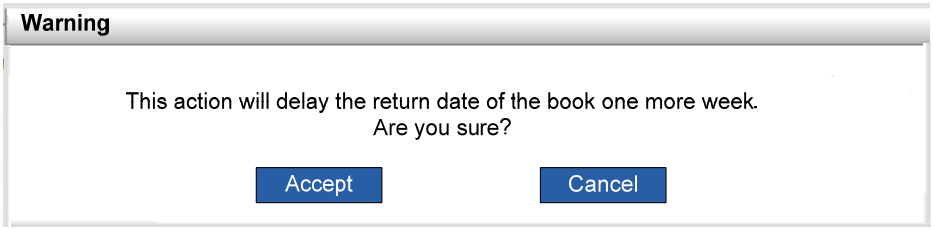


Figure 12 Mockup of a warning message

Test generation

Check warning bypassed

```
Open("http://library.upv.es") //Open the application
Type("id=username", "Pablo") //Authenticate with an existing user
Type("id=password", "apasd")
ClickAndWaitPageToLoad ("id= login")
ClickAndWaitPageToLoad ("id= myBorrowedBooks")
Click("id= renew0")
AssertElementPresent("id=dialog")
ClickAndWaitPageToLoad ("id=acceptWarning")
AssertTextPresent("Book renewal accepted")
AssertText("id=remainingDays0" , "8")
```

Check warning cancelled

```
Open("http://library.upv.es") //Open the application
Type("id=username", "Pablo") //Authenticate with an existing user
Type("id=password", "apasd")
ClickAndWaitPageToLoad ("id= login")
ClickAndWaitPageToLoad ("id= myBorrowedBooks")
Click("id= renew0")
AssertElementPresent("id=dialog")
ClickAndWaitPageToLoad ("id=cancelWarning")
AssertTextPresent("Book renewal cancelled")
AssertText("id=remainingDays0" , "1")
```

5.5 Structured Text Entry

In order to filter the search of a book, users can insert a rank of years in which the book was published. According to requirements, the years must be inserted with four digits. To avoid mistakes of users, the input widget for years must include a mask to guarantee that the user inserts four digits. The values for the usability properties of Structured Text Entry are shown in Table 6.

Table 6. Usability properties for Structured Text Entry

Structured Text Entry	
Property	**Value specified by the analyst in the library case study**
Widget selection	Two widgets where the rank of years must be inserted
Regular expression	The regular expression is ####, in other words, four integers.

Figure 9 shows a mockup of interface to look for a book. The widgets *published between* include a mask to guarantee that both years have four digits.

Test generation

Check widget set invalid

```
Open("http://library.upv.es") //Open the application
ClickAndWaitPageToLoad ("id=search") //Go to search
Type("id=publishedFrom", "20000")
Type("id=publishedTo", "2009")
ClickAndWaitPageToLoad ("id=doSearch") //Search
AssertTextPresent("Invalid from number. Must be 4 digits")
```

In order to pass all these tests, it is required to change some aspects of the OOWS conceptual model defined in Figure 7. The new OOWS models are shown in Figure 13. Usability Properties have been included by means of stereotypes (Progress bar, Cancel, Warning, Mask) and a new class (Favourites). This is where our approach provides the intended additional value by linking explicitly TDD with MDSD: once tests are written (and the requirement still not implemented), the required changes are incorporated in the model (instead of in the program code), making true the metaphor of working at the conceptual modelling level for software production purposes, while fully exploiting the principles of the TDD approaches.

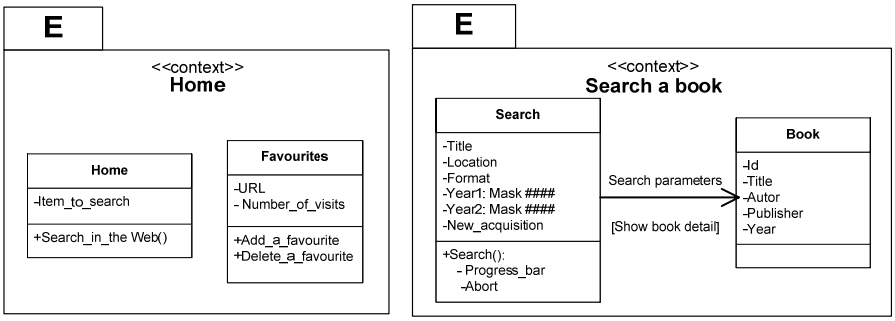


Figure 13a Modified OOWS model for home and search a book

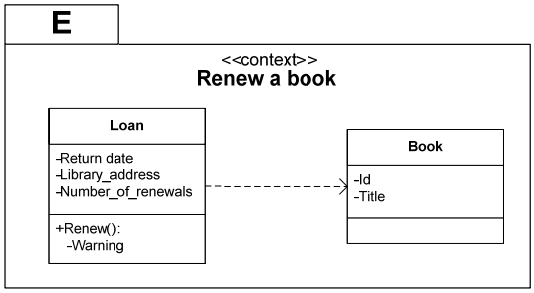


Figure 13b Modified OOWS model for renew a book

6 Concluding Remarks and Further Work

Recent studies have targeted the relationship between usability and functional requirements [2, 12]. We have presented a novel approach to include usability requirements strongly related to functionality

in a test driven, model-based development approach (MDSD). Usability properties are captured using a set of guidelines described in natural language. In order to fit this kind of requirements in the TDD cycle, we add tests that drive the development and check that the generated application is valid according to such requirements. The approach maintains the agile style while preserving an MDSD perspective, dealing with usability requirements in an incremental way. In order to exemplify our proposal, we have used a set of Functional Usability Features (FUF) for Web applications defined in the literature and we have explained how to define tests to validate those features in a specific MDSD method called OOWS.

Our proposal put together the advantages of agile methods and MDSD. On the one hand, all the software development process focuses on passing a set of tests defined with the help of the end user. That decreases possible misunderstanding between the end user and the analyst because the software can be validated quickly. On the other hand, the analyst concentrates all his/her efforts on building conceptual models, which are closer to the problem space than the implemented code. Additionally, a widely accepted software quality characteristic (usability) is incorporated to the proposed software production process from the requirements capture step. With all this work, it is our intention to demonstrate that:

i) Agile and MDSD can be adequately combined to reinforce each other, focusing on their respective good properties from a methodological perspective. We have shown that there are no contradictions associated to their combined use.

ii) Usability requirements can be incorporated to a MDSD method. Moreover, we have explained the advantages of dealing with usability from the early steps of the MDSD: less changes in the architecture design once the user sees the implemented system, usability can be included easily in the developing system by means of conceptual primitives.

We are currently working on several directions: First, we are working on an UID extension to easily derivate tests. As a proof of concept, we are developing a MDD tool that will simplify the process of UID construction and test generation. Second, we are doing some field experiences with usability requirements on RIA applications [10]. For this matter, we are analyzing how to validate those requirements in tests and where they should appear in the TDD cycle. Finally, in order to integrate all these features, we will extend the UID notation and tool to allow the specification of RIA properties.

Acknowledgements

This work has been developed with the support of MICINN under the project SESAMO (TIN2007-62894) and has been co-financed by ERDF. It also has the support of the Generalitat Valenciana by means of theORCA project(PROMETEO/2009/015).

References

1. Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development. http://www.agilemodeling.com
2. Bass, L., Bonnie, J.: Linking usability to software architecture patterns through general scenarios. The journal of systems and software 66 (2003) 187-197
3. Beck, K.: Test Driven Development: By Example (Addison-Wesley Signature Series), 2002
4. Bryc, R.: Automatic Generation of High Coverage Usability Tests. Conference on Human Factors in Computing Systems (CHI), Doctoral Consortium. ACM, Portland, USA (2005) 1108-1109
5. CARE: www.care-t.com. Last visit: October 2009

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 June (2000).
7. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishing, London (2000).
8. Cysneiros, L.M., Kushniruk, A.: Bringing Usability to the Early Stages of Software Development. International Requirements Engineering Conf. IEEE(2003) 359- 360
9. Dinh-Trong, T.T., Ghosh, S., France, R.B.: A Systematic Approach to Generate Inputs to Test UML Design Models. 17th International Symposium on Software Reliability Engineering (2006) 95-104
10. Duhl, J. Rich Internet Applications. A white paper sponsored by Macromedia and Intel, IDC Report, 2003
11. Estrada H., Martínez A., Pastor O. and Mylopoulos J., An empirical evaluation of the i* framework in a model-based software generation environment, CAISE 2006, Springer LNCS 4001, (2006) pp: 513-527.
12. Folmer, E., Bosch, J.: Architecting for usability: A Survey. Journal of Systems and Software, Vol. 70 (1) (2004) 61-78
13. Fons J., P.V., Albert M., and Pastor O: Development of Web Applications from Web Enhanced Conceptual Schemas. ER 2003, Vol. 2813. LNCS. Springer (2003) 232-245
14. Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.
15. Hailpern, B., Tarr, P.: Model-Driven Development: the Good, the Bad, and the Ugly. IBM Syst. J. 45 (2006) 451-461
16. IEEE Software, vol. 24, no. 3, May/June 2007.
17. Gómez, J. and Cachero, C. 2003. OO-H Method: extending UML to model web interfaces. In information Modeling For internet Applications, P. van Bommel, Ed. IGI Publishing, Hershey, PA, 144-173.
18. Griffiths, R.: The Brighton Usability Pattern Collection. http://www.cmis.brighton.ac.uk/research/patterns/home.html (2002)
19. Jacobson, I, Object-Oriented Software Engineering: A Use Case Driven Approach, ACM Press, Addison-Wesley, 1992.
20. Jacobson, I., Booch, G. and Rumbaugh, J (1999). The Unified Software Development Process
21. Jeffries, R. E., Anderson, A., and Hendrickson, C. 2000 Extreme Programming Installed. Addison-Wesley Longman Publishing Co., Inc.
22. Juristo, N., Moreno, A.M., Sánchez, M.I.: Guidelines for Eliciting Usability Functionalities. IEEE Transactions on Software Engineering, Vol. 33 (2007) 744-758
23. 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, 157-191. Springer (2008).
24. Lauesen, S.: Usability Requirements in a Tender Process. Computer Human Interaction Conference, 1998, Australia (1998) 114-121
25. Lawrence, B., Wiegers, K. and Ebert, C., The top risk of requirements engineering, IEEE Software, Vol. 18 (2001), pp: 62-63.

26. Maximilien, E. M. and Williams, L. 2003. Assessing test-driven development at IBM. In Proceedings of the 25th international Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 564-569.
27. MDA: http://www.omg.org/mda Last visit: October 2009.
28. Müller, M., Padberg, P. About the Return on Investment of Test-Driven Development. International Workshop on Economics-Driven (2003)
29. Nielsen, J.: Agile Usability: Best Practices for User Experience on Agile Development Projects. Nielsen Norman Group Report (2008)
30. Panach, J.I., España, S., Moreno, A., Pastor, Ó. Dealing with Usability in Model Transformation Technologies. ER 2008. Springer LNCS 5231, Barcelona (2008) 498-511
31. Pastor, O., Molina, J.: Model-Driven Architecture in Practice. Springer, Valencia (2007)
32. Perzel, K., Kane, D.: Usability Patterns for Applications on the World Wide Web. PloP'99 Conference (1999)
33. Rasmussen, J.: Introducing XP into Greenfield Projects: lessons learned. *IEEE Softw*, 20, 3 (May-June 2003) 21- 28
34. Robles Luna, E.; Grigera, J.; Rossi, G.: Bridging Test and Model Driven Approaches in Web Engineering. ICWE 2009.
35. Rossi, G., Schwabe, D.: Modeling and Implementing Web Applications using OOHDM. In Web Engineering, Modelling and Implementing Web Applications, 109-155. Springer (2008).
36. Selenium web application testing system. http://seleniumhq.org/
37. TestNG: http://testng.org/ Last visit: November 2009
38. Tidwell, J.: Designing Interfaces. O'Reilly Media (2005)
39. UML: http://www.uml.org/ Last visit: November 2009
40. Welie, M.v., Traetteberg, H.: Interaction Patterns in User Interfaces. 7th. Pattern Languages of Programs Conference, Illinois, USA (2000)
41. XML Path Language (XPath). http://www.w3.org/TR/xpath
42. Yu, E.: Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In: IEEE (ed.): IEEE Int. Symp. on Requirements Engineering (1997) 226-235
43. Watir: http://watir.com/ Last visit: November 2009
44. Wieczorek, S., Stefanescu, A., Fritzsche, M., Schnitter, J.: Enhancing test driven development with model based testing and performance analysis. Testing: Academic and Industrial Conf Practice and Research Techniques, TAIC PART '08 (2008)82-86.
45. Zhang, Y.: Test-driven modeling for model-driven development. IEEE Software 21 (2004) 80-86