



Repositorio Institucional de la Universidad Autónoma de Madrid

<https://repositorio.uam.es>

Esta es la **versión de autor** del artículo publicado en:

This is an **author produced version** of a paper published in:

Journal of Systems and Software 105 (2015): 107 – 124

DOI: <http://dx.doi.org/10.1016/j.jss.2015.04.023>

Copyright: © 2015 Elsevier

El acceso a la versión del editor puede requerir la suscripción del recurso

Access to the published version may require subscription

Design and Programming Patterns for Implementing Usability Functionalities in Web Applications

Francy D. Rodríguez¹

¹Escuela Técnica Superior de Ingenieros Informáticos, Universidad Politécnica de Madrid
Campus de Montegancedo s/n, 28660 Boadilla del Monte, Madrid, Spain
fd.rodriguez@alumnos.upm.es

Silvia T. Acuña²

²Departamento de Ingeniería Informática, Universidad Autónoma de Madrid
Calle Francisco Tomás y Valiente 11, 28049 Madrid, Spain
silvia.acunna@uam.es

Natalia Juristo¹

¹Escuela Técnica Superior de Ingenieros Informáticos, Universidad Politécnica de Madrid
Campus de Montegancedo s/n, 28660 Boadilla del Monte, Madrid, Spain
natalia@fi.upm.es

Abstract

Usability is a software system quality attribute. There are usability issues that have an impact not only on the user interface but also on the core functionality of applications. In this paper, three web applications were developed to discover patterns for implementing two usability functionalities with an impact on core functionality: Abort Operation and Progress Feedback. We applied an inductive process in order to identify reusable elements to implement the selected functionalities. For communication purposes, these elements are specified as design and programming patterns (PHP, VB .NET and Java). Another two web applications were developed in order to evaluate the patterns. The evaluation explores several issues such as ease of pattern understanding and ease of pattern use, as well as the final result of the applications.

We found that it is feasible to reuse the identified solutions specified as patterns. The results also show that usability functionalities have features, like the level of coupling with the application or the complexity of each component of the solution, that simplify or complicate their implementation. In this case, the Abort Operation functionality turned out to be more feasible to implement than the Progress Feedback functionality.

Keywords: Software Engineering; Design patterns; Programming Patterns; Usability; Abort Operation; Progress Feedback.

1. Introduction

Usability is a critical software system quality attribute in highly interactive systems (Juristo et al., 2007a). Usability is defined in ISO Standard 9241-210 (ISO, 2010) as “the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use”. SE originally considered that a satisfactory level of usability could be achieved by including usability features in the design of the user interface (UI). In this scheme of things, it was sufficient to use strategies that separated the UI from the core functionality of the applications.

It was later established that the separation strategy is not good enough to output a usable system, and there are usability issues that should be tackled as of the early development process activities (Juristo et al., 2007a) because they affect the core functionality of the applications. A usability

issue with impact on the software system core that is not taken into account early on in the development process will generate high costs, and the new system is unlikely to implement all its features (John et al., 2009).

The literature abounds with studies that deal with usability in early development process activities and present high-level solutions. Some of the proposals for including usability in software development are introduced as guidelines or patterns. For example, Juristo et al. (2007b) propose guidelines for eliciting requirements and Bass et al. (2001) and Folmer et al. (2003) introduce architectural patterns for including usability functionalities such as aggregating commands, cancelling commands, predicting task duration and verifying resources. Other approaches tackle activities later on in the development process. Thus, Juristo et al. (2007a) analyse the impact of usability issues on detailed design, and Folmer et al. (2006) present final implementations as an example to help establish the implications of usability for system architecture.

This research is a continuation of the effort to address usability issues that affect software system functionality within the SE development process. The difference is that it targets the later activities of the development process not normally addressed in the literature. We set out to establish whether it is possible to find reusable detailed design and programming solutions in order to build applications that implement usability functionalities. In this study we also analyse whether the identified reusable solutions can be specified as design patterns and programming patterns (D&P patterns). Finally, independent developers use the proposed solutions to implement systems for the purposes of evaluation.

We selected two usability functionalities called usability mechanisms (UM) (Juristo et al., 2007a), which have a major impact on design: Abort Operation (AO) and Progress Feedback (PF). These two functionalities cannot be implemented focusing on the UI only. The study is limited to web applications. Web applications differ from other application types in that the client side is composed of dynamic web pages which are interpreted by a browser and generate particular reuse conditions. A web page can be created on the server side or client side depending on the programming type or technologies used (W3C, 2014).

The object-oriented design and programming paradigm and three different server-side languages are used: PHP, VB.NET and Java. Object-oriented programming encourages reuse (Szyperski, 2002), and, as all three development projects use the same type of elements, we can look for the elements that they have in common. Although PHP was not originally an object-oriented language, its latest versions provide for the design and use of classes and methods. The web client side uses the Javascript language.

This paper was structured as follows. Section 2 presents the background and work related to our proposal. Section 3 describes the research method applied in order to both identify and evaluate the reusable elements. Section 4 shows the reusable elements discovered by the research and their specification as patterns for the AO and PF UMs. Section 5 describes how the proposed patterns were evaluated. Section 6 discusses the results and their evaluation. Section 7 presents the conclusions and future work.

2. Background

2.1 Usability mechanisms

The field of human-computer interaction (HCI) has addressed system usability at length. HCI guidelines are useful for achieving a satisfactory level of system usability. HCI researchers have defined a great many patterns bearing different names: interaction or interaction design patterns (Tidwell, 2010; Welie and Trætteberg, 2000), user interface patterns (Laakso, 2003), usability

patterns (Brighton, 1999; Perzel and Kane, 1999), and web design patterns (Van Duyne et al., 2006). All these patterns have in common that they offer solutions to specific usability problems, although they are described or grouped differently. There are also several pattern libraries for user interface design built by companies and available on the web (Yahoo, 2014; Pattern Factory Oy, 2014; Infragistics, 2015; Toxboe, 2015).

Based on HCI recommendations about how to improve software systems usability, Juristo et al. (2007a) identified three categories of recommendations depending on their effect on software development: usability recommendations with an impact on the UI, usability recommendations with an impact on the development process and usability recommendations with an impact on design. They reported empirical evidence of the relationship between usability and software design, identified functional usability features (FUF) with a high impact on design and measured their impact on real-world applications. The identified functionalities are a product of the HCI recommendations. In turn, each HCI author identifies different FUF subtypes. Each subtype has been referred to as UM and has a name indicating its functionality. A non-exhaustive list of FUFs and their respective mechanisms is presented in (Juristo, et al., 2007b). Table 1 shows the identified usability features and their respective mechanisms.

Table 1. Usability mechanisms with an impact on software design

Usability Feature	Usability Mechanism	Goal
Feedback	System Status	To inform users about the internal status of systems.
	Interaction	To inform users that the systems has registered a user interaction, i.e. that the system has heard the user.
	Warning	To inform users of any action with important consequences.
	Progress Feedback	To inform users that the system is processing an action that will take some time to complete.
Undo/Cancel	Global Undo	To undo system actions at several levels.
	Object-Specific Undo	To undo several actions on an object.
	Abort Operation	To cancel the execution of an action or the whole application.
	Go Back	To go back to a particular state in a command execution sequence.
User Input Error Prevention /Correction	Structured Text Entry	To help prevent the user from making data input errors.
Wizard	Step-by-Step Execution	To help users to do task that require different steps with user input and correct such input.
User Profile	Preferences	To record each user's options for using system functions.
	Personal Object Space	To record each user's options for using the system interface.
	Favourites	To record certain places of interest for the user.
Help	Multilevel Help	To provide different help levels for different users.
Command Aggregation	Command Aggregation	To express possible actions to be taken with the software through commands that can be built from smaller parts.

The use of the term usability functionality is potentially controversial, as usability is typically construed as being a non-functional requirement. However, Juristo et al. (2007b) established that the features listed in Table 1 “represent particular functionalities that can be built into a software system to increase usability. Since functional requirements describe the functions that the software is to execute, we consider that the usability features in Table 1 should be treated as functional requirements (even though they are usability-related requirements). Such functional usability requirements need to be explicitly specified, just like any other functionality”. Previous research by Bosch and Juristo (2003) and Bass et al. (2004) had already demonstrated the relationship between usability and software system functionalities.

In this paper, we propose D&P patterns to implement two of the UMs listed in Table 1. There are another two papers based on the usability functionalities and mechanisms described in Table 1. The aim of both papers is to add usability functionalities to software systems, but they take completely different approaches. One of the approaches (Carvajal et al., 2014) proposes guidelines for developers to incorporate FUFs into each development process activity from the requirements elicitation to the design stages. The second approach (Panach et al., 2014) is an extension of Juristo et al.'s research for model-driven development (MDD). Their aim is to build usability functionalities into software products developed using MDD.

We selected two UMs: Abort Operation, part of the Undo/Cancel FUF, and Progress Feedback, part of the Feedback FUF. Both UMs are highlighted in grey in Table 1. The other mechanisms belonging to these two FUFs are Global Undo, Object-Specific Undo and Go Back for the Undo/Cancel FUF, and System Status, Interaction and Warning for the Feedback FUF. The UMs were selected according to three criteria: impact on design in terms of number of affected functionalities, which was determined by the characteristics of the use cases to be developed; ease of recognition by a system user, and ease of evaluation from the viewpoint of HCI recommendations.

In order to apply the first criterion, UM impact on design, we analysed the requirements of the three systems under development and the respective elicitation guidelines in order to determine which UMs would be more useful for the new web applications. These are UMs that would need to be implemented more reliably at application development time. We established three possible values: low, medium and high. As regards the ease of recognition by system users, the analysis focused on the HCI recommendations associated with each UM in order to determine how many and what type of components the UM would have at UI level. Only the components whose functionality is executed by means of a UI element were selected. As regards the ease of evaluation from the viewpoint of HCI recommendations, we estimated the possible results of final implementations in order to establish whether it would be possible to recognize elements of the recommendations associated with each UM in the applications. The results of this analysis are shown in Table 2.

Table 2. Selection of UMs for implementation

UM	Level of impact on design	Ease of recognition by users	Ease of evaluation according to HCI recommendations
System Status	Low		
Interaction	High	X	X
Warning	Low		
Progress Feedback	Medium	X	X
Global Undo	Low	X	X
Object-Specific Undo	Low	X	X
Abort Operation	High	X	X
Go Back	Low	X	X
Structured Text Entry	High	X	
Step-by-Step Execution	Low	X	X
Preferences	High	X	X
Personal Object Space	Low	X	X
Favourites	Low	X	X
Multilevel Help	Low	X	X
Command Aggregation	Low	X	

2.2 Pattern-based reuse

In SE reuse aims to leverage the knowledge and experience gathered about software construction over time. Generally, it is about wrapping tested functionalities to be built into different systems, functionalities that have likewise been developed using SE methods. One of the benefits of reuse is

that it cuts costs and boosts productivity by using functionalities that have already been developed and tested, cutting software development times and maintenance costs, increasing reliability and reducing errors (Postmus and Meijler, 2008; Mellarkod et al., 2007).

There are different concepts and/or terms associated with reuse in the software development process, including libraries, toolkits, components, patterns and frameworks. There are overlaps between the above terms. For example, depending on the definition of component, a library may be classed as a component. A framework may use components and/or specific libraries. Also, frameworks may implement patterns, and components may be implemented using one or more patterns.

In this paper we have focused on two of these concepts, patterns and libraries. In object-oriented design and programming, libraries are a set of classes that cooperate with each other to achieve an aim. Microsoft foundation classes that cluster a lot of commonly used classes in a series of dynamic link libraries (dll) are one example. Libraries, which can be as complex as several operating system layers or very simple with basic utilities, have managed to increase productivity substantially (Goodliffe, 2006).

Patterns were originally reported by Christopher Alexander for civil engineering and architecture (Alexander, 1979; Alexander et al., 1997) and were later adopted by the software community after they were popularized by Gamma et al. (Gamma et al., 1997). There is no one generally accepted definition of pattern, but one of the simplest and most commonly used descriptions was given by Bushmann et al. (1996): "Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution". Bushmann et al. (1996) claim that "patterns act as building-blocks for constructing more complex designs. This method of using predefined design artifacts supports the speed and the quality of your design... Patterns help solve problems, but they do not provide complete solutions." In most circumstances, more than one pattern has to be used to complete a design. SE patterns are classed according to the software development process that they target. Accordingly, there are analysis patterns, architecture patterns, design patterns and programming patterns.

Design patterns deal with recurring software design problems that arise in particular situations. They are successful solutions to common problems. In the context of object-oriented programming, a design pattern is a description of classes and objects which communicate with each other to solve a general design problem in a particular context (Gamma et al., 1997). Design patterns provide a tested and documented solution to software development problems that are subject to similar contexts.

Programming patterns, also known as idioms, are low-level patterns that describe how to implement certain tasks using a particular programming language. These patterns entail the development of component parts or component relationships using language features (Bushman et al., 1996). A programming pattern is a known program element that can be used as part of the solution to many problems. Programming patterns delve into the implementation details using a specific language. They can offer a standalone solution or describe how to implement particular aspects of the design pattern components using the programming language features and resources. Each pattern can be named and renamed over again and can be rewritten several times based on user feedback. On this ground, pattern improvement is a continuous process, and they will evolve as technologies change.

When the level of abstraction of the design patterns is low, that is, they provide implementation details, they are called implementation-oriented design patterns or sometimes directly programming patterns. Examples are the singleton and the iterator patterns (Alur et al., 2003).

In this paper we present reusable solutions for the recurrent problem of implementing particular usability functionalities. We present low-level solutions, addressing detailed design and programming activities. We have specified the solutions as implementation-oriented design and programming patterns. We use a pattern template divided into sections using a naming convention that is widely recognized and used in SE: name, problem, context, solution, structure, implementation and example.

The solutions that we propose are compatible with the definition of pattern because they are not straightforward and have been successfully tried out on more than three systems. The best option for such solutions are implementation-oriented design patterns because they detail how to codify the design, thus providing helpful guidance and documentation for software developers. In this paper, we propose reusable low-level solutions for implementing the AO and PF UMs in web applications. Two implementation-oriented design patterns are defined, one for each UM. Based on the two implementation-oriented design patterns, we also present three programming patterns for each UM leveraging the features of the PHP, VB .NET and Java languages.

2.3 Patterns for SE usability

SE researchers have also conducted numerous studies and proposals for addressing usability using patterns. As already mentioned, SE originally considered usability as a feature associated exclusively with the UI, and therefore the developed solutions were consistent with the strategy of separating the UI from the core functionality of the application. Such solutions can use different interfaces for the same functionality and UI-level changes do not affect the application core. Examples of these solutions are the model-view-controller (MVC) and the presentation-abstraction-control (PAC) patterns. Later, however, the separation approach was found to be insufficient for implementing, debugging and maintaining some usability features (Bass and John, 2001).

The changes in how usability has been addressed in SE have led to the proposal and research of solutions that cover the entire software development cycle from requirements elicitation (Juristo, et al., 2007b), through architecture (Biel et al., 2011) (John et al., 2009) (John et al., 2004) (Bass and John, 2003) (Bass et al., 2001) and high-level design, to low-level design and implementation (Folmer et al., 2006). John et al. (2009) describe a study applying architecture patterns to support business-level usability. The results of this study offer a general description of what responsibilities the different functional elements must fulfil, but do not propose low-level solutions for implementing usability issues.

Bass et al. (Bass and John, 2003) (Bass et al., 2001) identified usability facets that require software architectural support rather than UI separation. The authors specify each facet of usability as a scenario with a characteristic stimulus and response. They provide an architectural pattern to implement each scenario. Examples of these scenarios include aggregating commands, cancelling commands and using applications concurrently. The abort operation mechanism defined in Juristo et al. (2007b) matches Bass and John's cancelling command scenario. As we explain later in the reported research, we define a set of application scenarios for each UM. We use the term scenario with a different meaning to Bass and John. Our scenarios refer to specific use cases generated by the UM implementation within applications. Although the notion of scenario differs from one paper to the other, the responsibilities of the architecture pattern components proposed by Bass et al. for the canceling commands scenario are equivalent in several respects to the responsibilities identified in this paper for the AO UM.

The STATUS project (STATUS Project, 2001) is in keeping with the above research. It examined the relationship between software architecture and usability and presented an approach for improving usability applying a specified design process. The STATUS project advocates that usability should be accounted for early on in the development process and brings forward the evaluation/improvement cycle to the system architecture phase. It proposes guidelines (Juristo et al.,

2007b) for eliciting usability functionalities prior to architecture definition. These guidelines are useful for adding usability functionalities from the very the first stage of the development process, namely requirements elicitation.

We find that hardly any of the above-mentioned HCI and SE patterns provide details on low-level software design or implementation. This means that architects and designers developing new software systems are not going to have enough information, particularly as regards usability issues that have an impact on software architecture and design, in order to evaluate the impact of building a usability feature into the system.

In this respect, Folmer et al. (2006) put forward the concept of bridge patterns as an extension of HCI patterns showing generic implementations for highlighting troublesome issues and their solutions. In order to describe these bridge patterns, they added two sections to the HCI pattern definition: architectural implications and an example of the specific implementation in terms of classes and objects and/or in terms of technologies or techniques used. Their aim was to help software architects to evaluate the implications of the pattern in their particular context and decide whether they need to modify the architecture to accommodate the pattern. They intended to provide an instrument for improving communication at the boundary between SE and HCI.

Our research follows Folmer et al.'s approach in that it provides real implementations, but, unlike Folmer, we set out not only to clarify for software architects the potential implications of usability functionality for systems architecture and design, but also provide an implementation-level solution that can be reused as both a low-level design pattern and a reusable code library.

3. Research approach

We used a three-stage inductive research method in order to tackle the first objective of searching for reusable elements to implement the AO and PF UMs, whereby we implemented case studies in order to induce a general solution. This solution was evaluated in a fourth phase (see Fig. 1). In the following we briefly summarize these phases, although the detailed description of their development is beyond the scope of this paper. Here we report the final results and their evaluation. These results include application scenarios, implementation-oriented design patterns, programming patterns and libraries.

Phase I. The usability functionality elicitation guidelines for the two selected UMs and a set of real requirements for three web applications constitute the input for this phase. During this phase we developed the three applications assuring the traceability of the elements related to the usability functionalities across the different development process activities: elicitation and specification, architecture and design, and implementation. In requirements elicitation and specification the functionalities related to each UM are marked with different font types. In design the components related to the UMs are marked with a different colour in both the class diagram and the sequence diagrams. Additionally, the code is carefully documented during implementation in order to single out code that is related to usability functionalities.

We developed three case studies which are all interactive web applications. The first is an indicator administration system designed to create simple indicators and data and classify, query and import data. The system was built in PHP 5 and has a MySQL database. The second case study is a web system for generating payment variables and can update and manage payroll information, calculating information on overtime, nights, weekends and work days. The system was built in Visual Basic .NET and has a Microsoft SQL database. The third case study is a healthy food electronic commerce system. It is a subscriber system that creates and maintains data on a subscriber's state of health, recommends a healthy diet, and provides several options for healthy food purchases and deliveries. The system was built in Java and has a Postgress database.

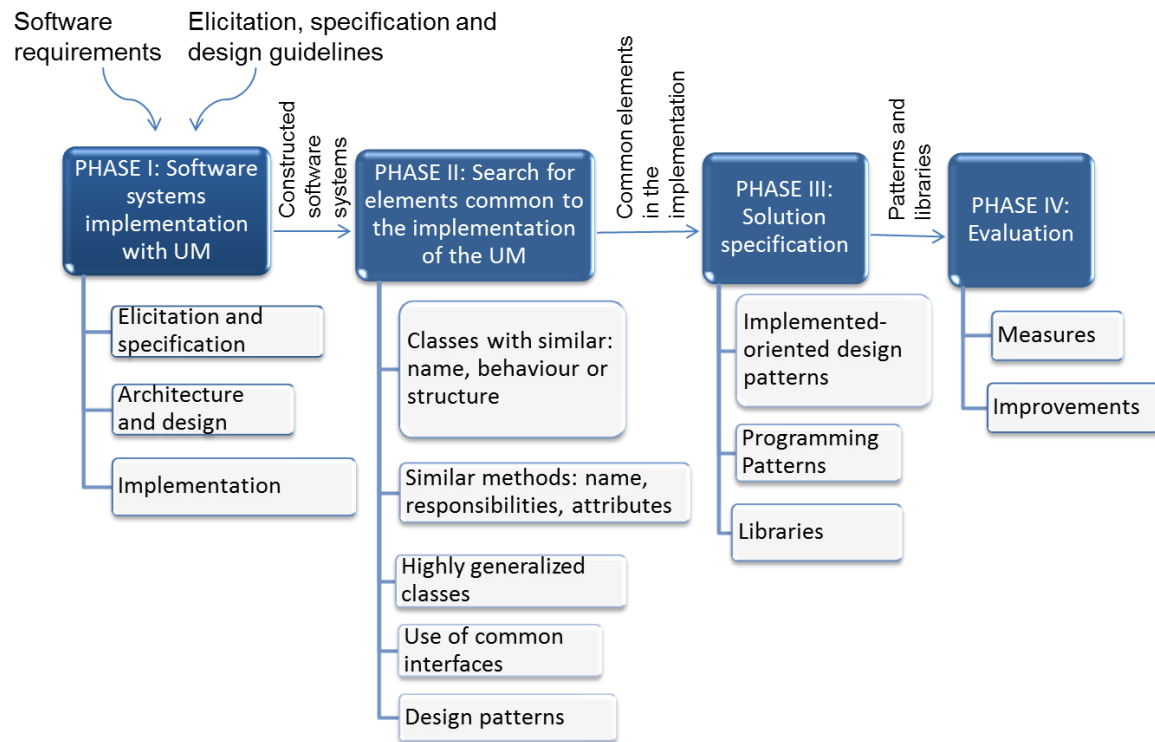


Fig. 1. Research method

Phase II. The three web applications with the two built-in UMs from Phase 1 are the input for this phase. In this phase, we are looking for elements that are common to all three implementations at the requirements definition, design decisions, responsibilities, classes, interactions, attributes and code level. The elements for implementing usability functionalities that are considered to be reusable constitute the output of this phase. As we are using object-oriented design and programming, the functionalities related to UMs are encapsulated in some classes of each application. Thanks to this we can establish the type of design used, single out the elements related to the UM and determine the type of interaction between the functionalities of each UM and the core functionalities of the applications.

Phase III. The common and reusable elements discovered in Phase 2 are the input for this phase. During this phase these elements are specified as implementation-oriented design and programming patterns. The aim behind this specification is to efficiently communicate the results for use in other development projects. Another two reusable elements are output on top of the patterns: a set of application scenarios and code snippets that are wrapped as libraries for each UM. The application scenarios represent all cases related to the usability functionalities across all three of the developed applications. These cases are the result of combining the questions stated in the usability functionality elicitation guidelines with their possible responses. Responses depend on the requirements of each application and, in some cases, the technology to be used.

Phase IV. We use a case study methodology (Runeson and Höst, 2009) for the second objective of validating the solutions. The proposed solutions are evaluated for exploratory purposes. The aim is to discover what happens when using implementation-oriented design patterns and programming patterns to develop web applications in order to implement the AO and PF Ums. The case studies evaluate three key issues: ease of understanding of each pattern, ease of use of each pattern and the impact on the developed web application.

The first issue, ease of pattern understanding, aims to determine what problems or obstacles developers face in order to comprehend the patterns, which additional activities or mechanisms are necessary in order to comprehend the patterns and how their ease of understanding might be improved. Another aim is to establish a measure of the time it takes for developers to understand the patterns.

The second issue, ease of use of each pattern, aims to determine what problems or obstacles developers face when they use the patterns, what facilities there are for pattern use, and which problem-solving artefacts they really use and in which development process activities. It also defines whether other artefacts need to be created and/or the solution needs to be modified in order to be used effectively, as well as how the ease of pattern use might be improved. Another aim is to measure the time taken to implement the pattern, the perceived difficulty and the impact of the usability functionality on the final application.

The third issue, final result, aims to establish whether the final applications include the specified usability functionalities. We also determine whether the developers find the patterns useful and whether they would use them in future developments. Another aim is to measure the complexity of the solution. The evaluation of these three issues is detailed in Section 5.

The problem context is highly interactive web applications developed using the object-oriented paradigm. We asked two independent developers to implement two web applications (units of analysis) based on real requirements and including functionalities associated with the two proposed UMs. The results of the first part of the research (patterns) were used for the purposes of evaluation. So, developers were given a document containing the implementation-oriented design pattern and programming patterns for the language that they selected for each UM. The document also contained the functionality elicitation guidelines for each UM and the application scenarios. They were also given reusable code wrapped up in libraries.

The independent developers had programming experience. The developers built the case studies as part of their Madrid Technical University (UPM) master's theses. One of the developers holds a BS in Computer Science and Engineering, a MS in Computer Science and Engineering and is taking the UPM's MS in Information Technologies, has five years' professional experience in software programming and design, is familiar with Visual Basic, Visual Basic .Net, Java, TeamUp, Javascript, MatLab, HTML and XSLT and was acquainted with the concept of usability before starting the case study. The other developer holds a BS in Computer Systems Analysis and is taking a BS in Computer Science and Engineering and the UPM's MS in Software and Systems, has four years of professional experience in programming and two years in software design, is fluent in Java, PHP and Visual FoxPro, acquainted with Visual Basic, C#, C, C++, Javascript and Perl, and unfamiliar with the concept of usability. Neither of the developers had previous experience in the use of design patterns and only one of them had used programming patterns.

One of the applications is an office supplies order control system for a nationwide company with offices in a number of cities around the country. The primary goal is to automate the office supplies query, order and reception system. The developer was given a preliminary requirements document containing 13 functionalities. The second application study is a software project requirements administration system. The system is able to define projects, make requests, specify and monitor requirements and administer the related documents. The goal is to improve communication between project team members and with customers. The developer was given a preliminary requirements document containing 14 functionalities.

The developers used different programming languages and development models. One of them used the Visual Basic .Net language and the incremental development model, whereas the other used the Java language and the waterfall model. Each application was developed over six months. Each developer met with the user and the researcher several times. Meetings were audio recorded. At the

meetings with the researcher, the developers were allowed to ask anything they liked about the use of the elicitation guidelines, scenarios and/or patterns. Developers also met another researcher who evaluated progress and advised on the process. At these meetings the principal investigator acted primarily as an observer.

4. Design and programming (D&P) patterns

During the first part of the research we managed to find common elements regarding the AO and PF UMs across all three developed web applications. We found that the description of the usability functionalities in the elicitation guidelines (Juristo, et al., 2007b) is still too general for the purposes of implementation. Analysing the requirements after implementing the usability functionalities, we observe that the functionality of each UM could be decomposed into more detailed application scenarios. Each combination of responses to the elicitation guideline questions generates an application scenario that may or may not be applicable depending on the software system in question. Trees with the identified combinations were built to give an overview of the discovered scenarios. Each scenario has a name identifying its functionality and is described by sequence diagrams.

The analysis of the classes and responsibilities reveals that class designs cover similar responsibilities for each UM in all three developments. We find that the extracted responsibilities overlap or are complementary, thereby providing a set of general responsibilities that cover the UM functionality across all three applications. We can then extract a general design based on the design of the three applications. The generic designs are specified as design or more specifically implementation-oriented design patterns.

After we had defined the implementation-oriented design patterns, we conducted a backward analysis to adapt the design and the code of each developed application to the proposed design pattern. The aim was to cluster the functionalities, design and code according to the design pattern components. The outcome was a programming pattern for each of the languages used for the two UMs. The end result was two implementation-oriented design patterns and three programming patterns for each UM.

Because the implemented systems are web applications, the results at programming level were different for the client side and the server side. Whereas the server-side design and code is very much influenced by the features of the language used —PHP, VB .NET and Java—, leading to variations not only in the code but also in the design, the only language used on the client side is Javascript, and we were able to unify the code for all three applications into a single client-side code for all three applications. This code was wrapped up in a single library for each UM.

In the following we show for each UM the scenarios, responsibilities, classes that meet these responsibilities, referred to generically as components, and the common design specified as a pattern. We will also describe some of the features of the programming patterns.


4.1. Abort Operation UM

The AO UM focuses on enabling the user to cancel an operation, a command or exit the application in a safe and predictable manner. The elicitation guideline for the AO UM divides the questions into three levels: application, operation and command. At the application level, the guideline indicates that users should be asked whether an option for exiting the application is necessary and, if it is, how the option will be displayed to the user. According to the HCI recommendation associated with the elicitation question, the option to quit must be immediately and obviously available, even if modal dialogues are used. If the quit option is selected after data have been modified, the save option must be displayed. The operation level refers to actions that involve the execution of one or more steps within an application, each of which requires interaction with the

user. Each action has the effect of changing the state of the application, either by modifying database information, changing configuration parameters or altering application or session variables for web applications. Finally, the command level refers to an instruction or order that the user gives the application by means of a single interaction, that is, pressing a button, clicking on a link, selecting a menu item or any other option offered by the application.

We analysed the application requirements specifications and found that the AO UM functionality is coupled with application functionalities. Table 3 shows a requirement specification for one of the systems, where item 6.1 under Alternative Paths denotes functionality associated with the AO UM. This is the same behaviour as is observed for all the requirements of all three applications. We find that the UM elicitation guideline questions used to elicit all the UM-related functionalities generate a large number of cases or scenarios.

Table 3. Example requirement specification with UM-related functionalities.

Identifier: CU26	Essential/Desirable: Essential	Priority: High
Use Case Name: Import indicator data		
Author/Modifier: Francy Rodríguez		
Date: 21-03-2009		
Category (Visible/Invisible): Visible		Actors: Administrator, user
Summary: Allows an application administrator or user to import indicator data from a flat file		
Preconditions: Valid user on the system who has the necessary permits to execute the operation.		
Postconditions: Flat file data are saved to system, if correct.		
Basic Event Sequence		
User	System	
1. The actor selects the import option from the menu on the left of the navigation window which pops up after pressing the Administration option.	2. The system opens a page to which the information to be imported can be copied.	
3. The actor copies the information to be imported.	5. The system starts to import data.	
4. The actor presses the Process option	6. The system displays a modal window with a progress bar and cancel button.	
	7. Import ends and the system displays an “operation successfully completed” message.	
Alternative Paths		
6.1 <i>The actor presses the modal window Cancel option, the system does not save changes and reverts to the previous state.</i>		
Exception Paths		
5.1 If data import errors are detected, the system displays an error message and reverts to the previous state.		
Extension Points		
Draft graphical user interface		
<p>Paste data of flat file and press process button</p> 		

From the analysis of all the scenarios, it was found that there were four issues that together generated the different cases: whether or not there were changes to be saved when the user uses the AO functionality, whether or not the user wants to save those changes, whether or not the changes are successfully saved and the source of the request. We identified a total of 22 AO UM scenarios, of which were 16 operation-level scenarios, five were application-level scenarios and one was a command-level scenario. The difference between the number of operation- and application-level

scenarios is due to the fact that there are four possible sources at operation level (dialogue box containing a Cancel button, form containing a Cancel button, selection of another application option and Clear button) rather than just the one at application level (Exit option).

Fig. 2 shows the operation-level scenario tree. The elements represented in rounded rectangles denote the four issues which together generate the different cases. The elements illustrated in sharp-angled rectangles denote the messages and final application states in each case. Fig. 2 highlights the FormCancelButtonSavedChanges scenario, where the source of the Cancel operation is a button on a form, there are unsaved changes, the changes are to be saved and the changes are successfully saved.

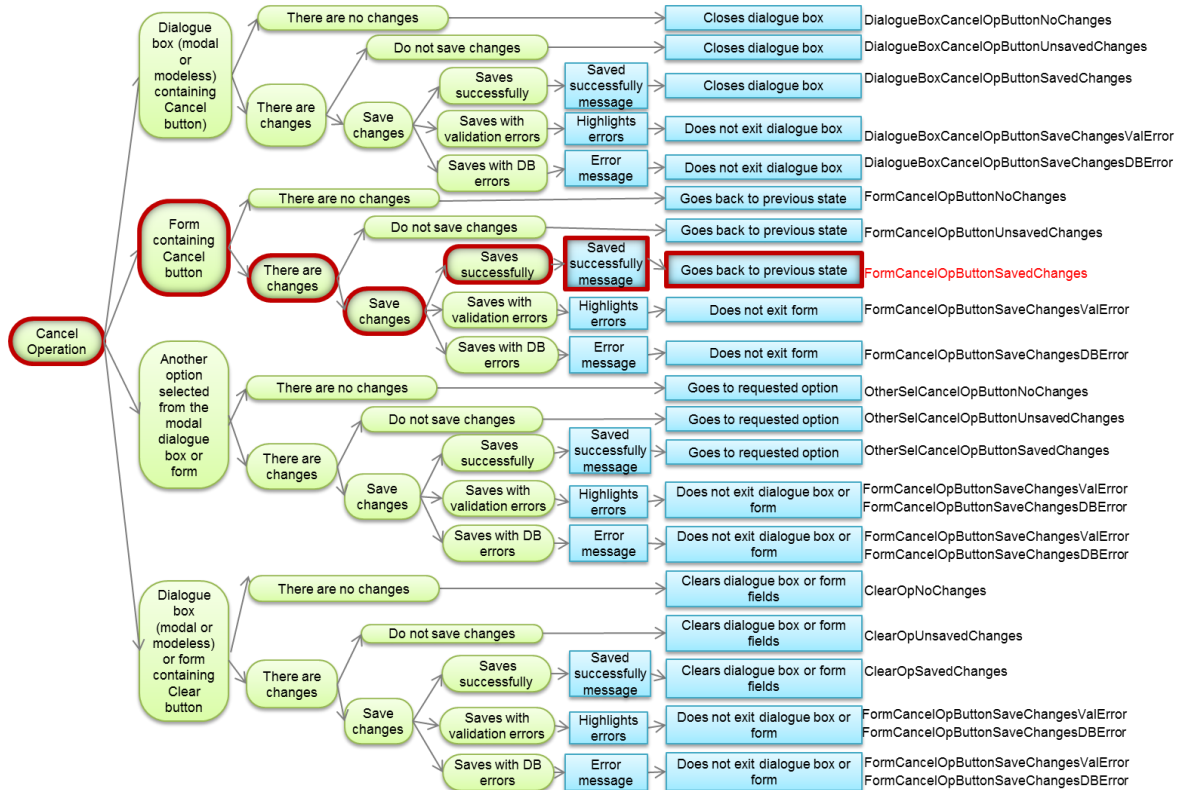


Fig. 2. Operational-level scenarios tree for the AO UM

The identified responsibilities for a system to meet the recommendations associated with the AO UM are:

- Listen to the user actions to determine when to quit the application, cancel an operation or cancel a command.
- Know whether or not there are changes to be saved at any time.
- If there are changes to be saved, ask the user whether or not to save these changes and know which action to take depending on the user response.
- Know the previous and current state of the application.
- Know how to save changes irrespective of the operation or command that is being executed.

The identified components for meeting the identified responsibilities are:

- ChangesChecker, which updates and reports changes to be saved in the application.
- CancelHandler, which saves changes if operations are aborted and gets the system into a state that is predictable and safe for users.

- UndoCancelFUF, which receives requests to abort operation (quit or cancel), asks the ChangesChecker component if there are any changes, asks users if they want to save changes and calls the respective method.
- StepHistory, which updates and provides information on the previous and current states of the application.

The interaction between components is modelled based on the application scenarios identified for the AO UM. Each scenario is described by means of a sequence diagram. Fig. 3 shows the sequence diagram for the FormCancelOpButtonSavedChanges scenario, which is highlighted in the tree in Fig. 2.

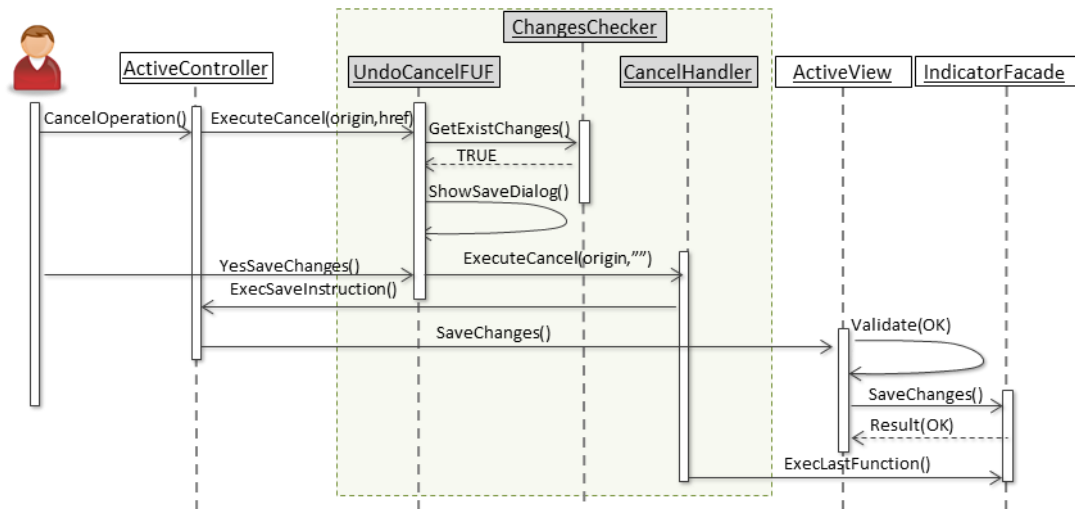


Fig. 3. Sequence diagram for FormCancelOpButtonSavedChanges scenario

The resulting design for the AO UM is shown in Table 4 under the structure heading. At attribute level, we found that although their definition varies depending on the technology used, it is also consistent on several points. For example, an attribute in the class that encapsulates the main methods for dealing with an abort operation request is used in all three cases to store the necessary information for saving the changes, although it is implemented differently in each case. The same applies to the instruction for closing a dialogue box or quitting the application.

Table 4 shows the unified design proposed for the Abort Operation UM specified as a pattern. The pattern template has different sections: name, target problem description and context, solution, structure, implementation and related patterns. The solution section details the responsibilities to be fulfilled by the usability functionality. The structure section includes the proposed design. As this is an implementation-oriented design pattern, it includes an implementation section that specifies the steps necessary to codify the proposed design.

Programming patterns differ from design patterns with respect to two key aspects. First, the design provided in the programming pattern may vary slightly from the structure proposed in the design pattern because programming language assets can be leveraged making it unnecessary to use all the components of the original design. In order to illustrate these differences, Fig. 4 shows the programming pattern design for AO UM in VB .Net. Compared to the design proposed under the structure heading in Table 4, no StepHistory class is necessary because the language itself provides the information on the previous application state. For the design of each programming pattern and their differences from the design proposed in Table 4, see the full documentation of each programming pattern for the AO UM in the Java, PHP and VB .NET languages available at http://www.grise.upm.es/sites/extras/7/PP_AO_Java.pdf, http://www.grise.upm.es/sites/extras/7/PP_AO_PHP.pdf and http://www.grise.upm.es/sites/extras/7/PP_AO_VB_NET.pdf, respectively.

Table 4. Abort Operation UM design pattern.

NAME	Abort Operation UM
PROBLEM	The user must be able to exit an application, operation or command immediately and quickly.
CONTEXT	Highly interactive web applications
<p>SOLUTION</p> <p>Components are required to fulfil the responsibilities associated with the UM. They are:</p> <ul style="list-style-type: none"> • A component to update and report on whether there are any changes to be saved in the application. • A component that queries whether there any changes to be saved and asks the user whether to save the changes after an abort operation request. • A component that knows everything it needs to know in order to save the changes, if any, after an operation is aborted. • A component that knows the next application state after an operation is aborted irrespective of whether or not there are any changes and whether or not they are to be saved. • A component that knows what the previous system state was. 	
<p>STRUCTURE</p> <pre> classDiagram class CancelHandler { +closeInstruction +saveInstruction +ExecuteCancel() +ExecuteClean() +ExecuteClose() +GetCloseInstruction() +GetSaveInstruction() +SetCloseInstruction() +SetSaveInstruction() } class UndoCancelFUF { +Changed() +ExecuteCancel() +ExecuteClean() +ExistChanges() +ResetChanges() +SetCloseInstruction() +SetCurrentStep() +SetSaveInstruction() } class ChangesChecker { +existChanges +Changed() +GetExistChanges() +Reset() } class StepHistory { +currentStep +lastStep +SetCurrentStep() +SetLastStep() } class OriginCancelType { None ButtonCancelForm ButtonCancelDialog ButtonClean Link } UndoCancelFUF --> CancelHandler : cancel UndoCancelFUF --> ChangesChecker : changes UndoCancelFUF --> StepHistory : history UndoCancelFUF --> OriginCancelType </pre>	
<p>IMPLEMENTATION</p> <ol style="list-style-type: none"> 1. Create a singleton UndoCancelFUF class. 2. Create a ChangesChecker that updates and provides information on application changes. 3. Create a StepHistory class that updates and provides information on the previous system state. 4. Create a CancelHandler class that knows how to save operations, clear fields and close dialogues, and which is the next system state after an operation is aborted. 5. Implement the UndoCancelFUF class methods to operate as a façade for the CancelHandler, StepHistory and ChangesChecker classes. 6. Implement the right functionality in each changeable part of the application (controllers for MVC) so that the state of ChangesChecker is updated if anything in the application is changed. 7. Implement the right functionality so that the system always knows which method to use or which action to take to save a change if an operation is cancelled or the application is quit. This can be done using the CancelHandler class. 8. Implement the right functionality so that the system knows which method to use or which action to take at any time in order to close a dialogue box, if any. This can be done using the CancelHandler class. 9. Implement the right functionality so that the system knows how to clear form fields or active dialogue boxes at any time. This can be done using the CancelHandler class. 10. Implement the right system functionality to save the latest state during application navigation so that this data item is available if it is necessary to restore a previous state. This can be saved in the StepHistory class. 	

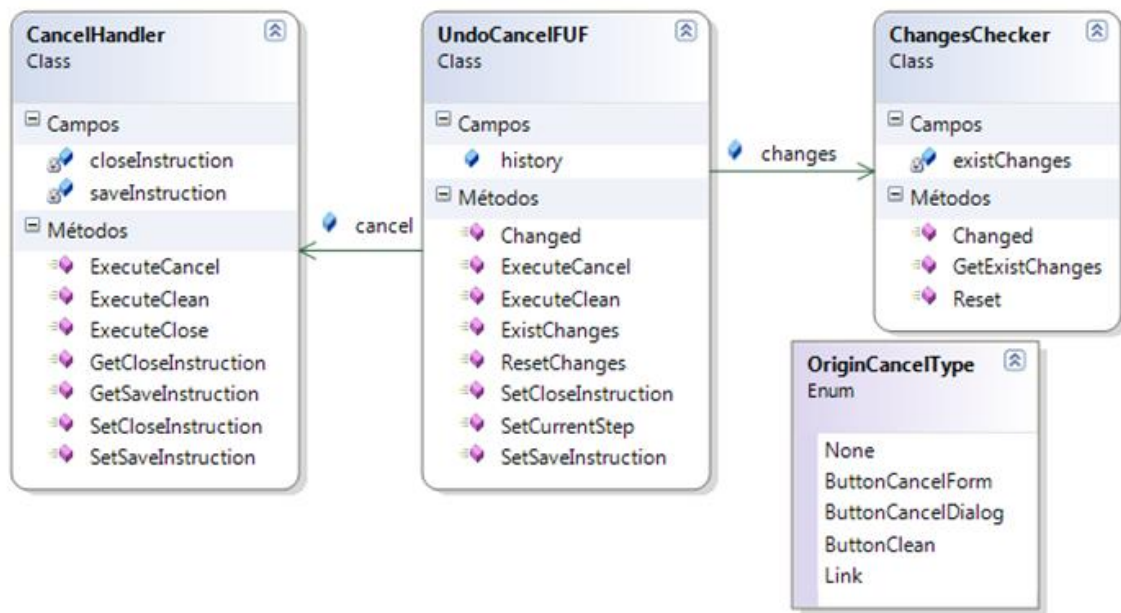


Fig. 4. Design of programming pattern for AO UM in VB .Net

The second distinctive feature of the programming pattern is that it has an additional section named example, which illustrates the code associated with each item described in the design pattern implementation section. The programming pattern example section lists the items of the implementation-oriented design pattern implementation one by one, together with their associated code and comments or examples to help programmers understand the solution. Table 5 shows an excerpt from the programming pattern example section for the Java AO UM. Note that this excerpt refers to item 7 of the design pattern implementation section and illustrates the documented code for different possible scenarios.

Table 5. Excerpt from the Java AO UM programming pattern example section

<p>EXAMPLE</p> <p>7. (...)Implement the right functionality so that the system always knows which method to use or which action to take to save a change if an operation is cancelled or the application is quit. This can be done using the CancelHandler class. <i>//Examples are presented below for each of the specified cases.</i></p> <p>FORM CONTAINING CANCEL BUTTON CASE</p> <p><i>// 7.1. On each page where information can be changed or created and cancelled, define an element that knows how to save the changes and go back to the previous state. In the example below, the method to be executed is updateCancel and is associated with a hidden commandLink:</i></p> <pre><h:commandLink id="updateCancel" style="visibility: hidden" action="#{clienteController.updateCancel}"/></pre> <p><i>//This method is different to the normal update method, because it always goes back to the previous state and consults the StepHistoryContext bean to find out which the previous state was (see 7.4)</i></p> <p><i>//7.2. Save the element id as a save instruction. When the application goes back to the previous state and has to save changes, it has to call the saved click event element. In this example the element was created in step 7.1 and its use is illustrated in the CancelHandler class, ExecuteCancel event, origin= FormCancelOpButton.</i></p> <pre>undoCancelFUF.SetSaveInstruction({buttonCancel: 'updateCancel', buttonClean: 'updateClean', saveChanges: 'saveChanges'});</pre> <p><i>//The saveInstruction attribute is an array of values, and the value is CancelButton in this case. The other values are explained in the following steps</i></p> <p><i>//7.3. Now create a cancel button, which, in the onclick event, calls to the ExecuteCancel function of the UndoCancelFUF façade with the respective parameters: the first parameter is type, which is in this case is a button on a form (FormCancelButton), and the second parameter is null because it is not required.</i></p> <pre><h:commandButton type="button" value="Cancel" onclick="JavaScript:undoCancelFUF.ExecuteCancel (OriginCancelType.ButtonCancelForm, null); " /></pre> <p><i>(...)</i></p>
--

4.2. Progress Feedback UM

The PF UM informs the user either graphically or textually of the progress of a process. As regards context, the PF functionality should be implemented when a process executing within an application is likely to block the UI for longer than two seconds. According to the elicitation guideline, the questions to be asked are: Which tasks are likely to take longer than two seconds? Which of the identified tasks are critical? How will the user be informed that the process has finished? How will the user be informed about the progress of each task? And what information is necessary in each case?

As for the AO UM, the UM-related functionalities are coupled with application functionalities. Table 3 shows that item 6 on the system side of the Basic Event Sequence denotes functionality associated with the PF UM. Such UM-related functionalities are defined in many other application requirements. Unlike the AO UM functionalities, which are always found in the alternative paths, the PF UM functionalities always appear in the basic event sequence.

We found that there are 12 application scenarios for the progress feedback functionality. The whole tree is shown in Fig. 5. The scenarios are conditioned by the possible responses to the elicitation questions and by the type of technology to be used. As in Fig. 2, the elements represented in rounded rectangles denote the issues which together generate the different cases. Unlike AO UM, however, the elements represented in sharp-angled rectangles specify whether or not the programming language handles multithreading. Fig. 5 highlights the scenario called `MultithreadedPIw/Infow/oCancelw/MSG`, which occurs if a process with progress information cannot be cancelled, must display a message for users upon process completion and uses multithreaded technology.



Fig. 5. Progress Feedback UM application scenarios

The responsibilities identified for the PF UM are:

- If progress information is available and the technology is multithreaded, determine whether a process is still active.
- Generate a server-side mechanism for the active process to update and report progress.
- Create a cyclical process that queries the progress of a task until completion.
- Display the right progress indicator depending on the available information.
- Inform the user of task completion.
- Display the completion message and close the progress indicator.

Five components were defined to fulfil these responsibilities:

- **ProgressFeedbackUI.** This component displays the right progress indicator depending on the available information —time, percentage, processed units, tasks completed—, or an indeterminate progress indicator when no information is available. It paints the progress indicator on the UI according to the parameters that it is given: title, size, process name, task name, modal or modeless, initial value, etc. It changes the values displayed at any time. It can reposition the progress indicator on the UI. It informs the user of process completion as instructed. It displays the Close or Cancel button and a completion message when necessary. It closes the progress indicator.
- **ProcessChecker.** This component is able to determine whether a process is still active. It establishes whether or not the progress indicator should still be displayed and checks its progress.
- **ProgressFeedbackHandler.** This component handles the user-generated events and server responses. It launches the right options depending on the event and the information it receives. It also accounts for the possibility of there being more than one progress indicator active at the same time. It is responsible for creating and updating the ProgressFeedbackUI class instances in order to display and update the information on screen. It manages cyclical processes that query a server object progress value every x units of time.
- **ProgressResult.** This is the server-side component that maintains the process progress information per session. Its function is to update and provide the process progress information when requested.
- **FeedbackFUF.** This component is a class that is used as a façade between the system and progress feedback functionalities. Its responsibility is to distribute the requests to usability functionality components reducing dependence on the application functionality.

In order to describe the interaction between these components, a sequence diagram is built for each scenario of the scenarios tree. Fig. 6 shows the sequence diagram for the scenario highlighted in the tree shown in Fig. 5. The final design proposed as a pattern for the PF UM and its specification as an implementation-oriented design pattern is shown in Table 6.

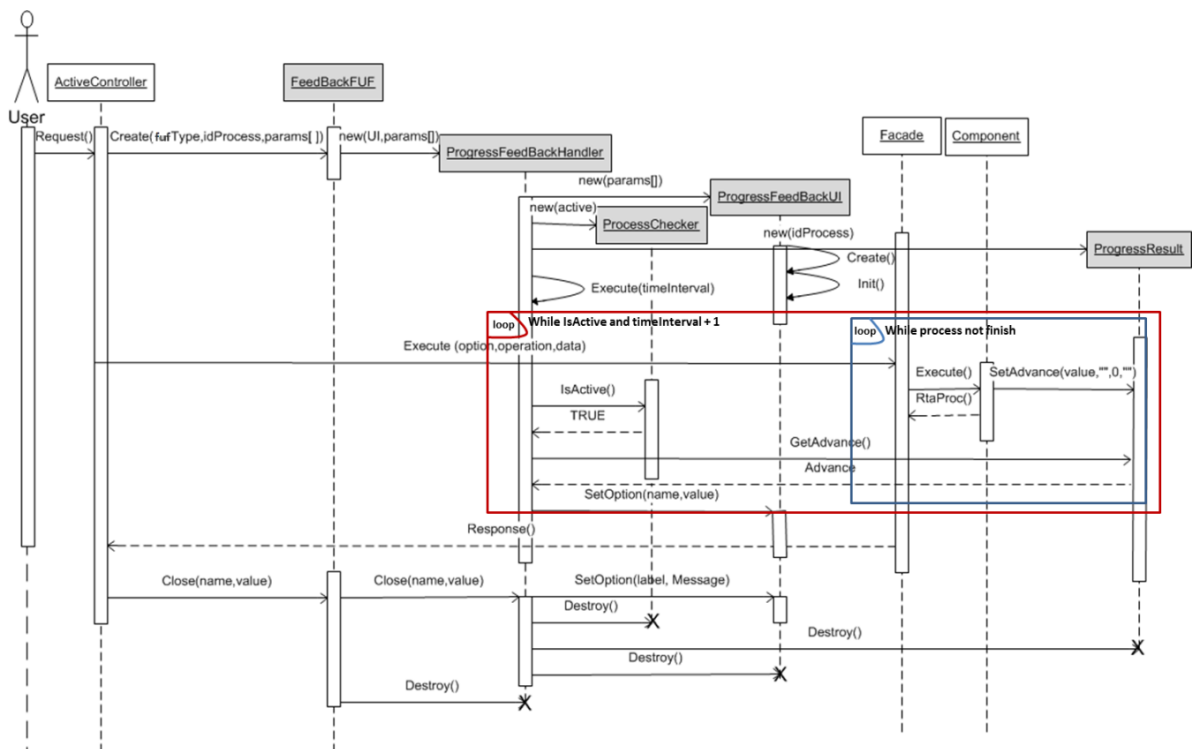


Fig. 6. Sequence diagram for MultithreadedPIw/Infow/oCancelw/MSG

Table 6. PF UM design pattern

NAME	Progress Feedback UM
PROBLEM	When a system process is likely to block the UI for longer than two seconds, the user must be informed about the progress of the requested task.
CONTEXT	Highly interactive web applications and time-consuming processes that interrupt the UI for longer than two seconds
SOLUTION	
<p>Components are required to fulfil the responsibilities associated with the UM. They are:</p> <ul style="list-style-type: none"> • A component that displays the right progress indicator depending on the available information: time, percentage, number of units processed, task completed or indeterminate progress indicator (if no quantities are known). • Generate a server-side mechanism for the active process to update and report progress. • Create a cyclical process that queries the progress of a task until completion. • A component to inform the user of task completion. • A component that displays the completion message and closes the progress indicator. • If progress information is available and the technology is multithreaded, determine whether a process is still active. 	
STRUCTURE	
IMPLEMENTATION	
<ul style="list-style-type: none"> • Create a ProgressFeedbackUI component as a widget to display the progress indicator and the user warning. A widget or control is an element of the graphical user interface (GUI) that displays information; it can run as a small standalone program but functions in this case as a process within the same application. • Create a ProgressFeedbackHandler class that acts as the handler of the main UM functions: periodically check the progress of a task or process to a component on the server (multithreaded systems) or may be informed from a client component forward (single-threaded systems), update the progress indicator and close the progress indicator when it is necessary. • If progress information can be gathered on the process while it is running on the server (multithreaded systems), create a ProgressResult class on the server side to which the process will report its progress and which can be queried from the client. • Create a CheckerProcess class that stores and provides information on whether a process is active. • Create a FeedbackFUF component that operates like a façade for all the components that are used in the progress feedback FUF functionality implementation: ProgressFeedbackHandler, ProgressFeedbackUI, CheckerProcess and ProgressResult. • Implement, through the Create () method of the FeedbackFUF façade, the right functionality in the application so that, for each process that takes more than two seconds, the progress indicator starts with appropriate parameters. • Implement the right functionality in the application so that, in multithreaded systems, the ProgressResult component will be updated with the right progress information and, in single-threaded systems, calls will be made to the server so that progress information can be supplied through the FeedbackFUF façade. • Implement the right functionality in order to inform through the FeedbackFUF façade when a process is cancelled in order to update the respective information in the widget (cancelling message) and also deactivate the process using the ProcessChecker class. • Implement the right functionality in order to close the progress indicator and send a notice or message when necessary. 	
RELATED PATTERNS	Singleton Pattern, Façade Pattern and MVC Pattern.

Table 7 is an excerpt from the PHP PF UM programming pattern example section. Note that this is the client-side code and is therefore written in Javascript. The full programming pattern for the PHP PF UM is available at http://www.grise.upm.es/sites/extras/7/PP_PF_PHP.pdf, and the VB .NET and Java patterns are accessible at http://www.grise.upm.es/sites/extras/7/PP_PF_VB_NET.pdf and http://www.grise.upm.es/sites/extras/7/PP_PF_Java.pdf, respectively.

Table 7. Excerpt from the PHP PF UM programming pattern example section

<p>EXAMPLE Create a ProgressFeedbackUI component as a widget to display the progress indicator and the user warning.</p> <pre> /* ----- //Widget: ProgressFeedbackUI -----*/ (function(\$, undefined) { \$.widget("ui.ProgressFeedbackUI", { options: { width: "300px", modal: false, name: null, labelTop: null, labelBottom: null, labelMin: null, labelMax: null, labelValue: null, viewCancel: false, autoClose: false, align: {}, value: 0 }, _create: function() { //Create progress feedback this.element .addClass('ui-progress-feedback') .attr({role: 'progressFeedback'}); this.\$dialogBox = \$('<div class="ui-progress-feedback-box ui-widget ui-widget- content ui-corner-all"></div>').appendTo(this.element); //Addheader . . (...) </pre>

5. Applying D&P patterns in case studies: opinion of developers

This section reports the results of the second part of the research concerning the evaluation of the use of the proposed solution by independent developers. As mentioned in Section 3, we evaluated three issues: ease of understanding of each pattern, ease of use of each pattern and impact on the developed web application. Each developer was to use the reusable elements that they were given and document the whole process in response to questions or taking the measurements set out in Table 8. The key findings from the analysis of developer responses are highlighted below.

5.1. Ease of understanding

As regards the first issue, the first noteworthy point is that the two developers had different impressions with respect to the ease of understanding of the usability functionality elicitation guidelines. Whereas one of the developers had no difficulty in applying the guidelines, the other considered that he had not received enough information about how to apply the guidelines. This is an important concern as regards the documentation of the entire process. The results of the analysis of the responses associated with this issue range from unclear explanations in the documents to the usual problems with applying new concepts like the pattern or new technologies like *jQuery*. The developers had to ask for more help from the researcher, thoroughly swot up on the delivered artefacts and look into additional issues unaided, all of which was very time consuming.

Table 8. Issues, questions and measurements for evaluating the proposed patterns

Issues for evaluation	Questions and measurements
Issue 1 (I1): Ease of pattern understanding	Q1.1: What problems or obstacles did you encounter when trying to understand the pattern? Q1.2: What additional mechanisms or activities did you need in order to understand the pattern? Q1.3: What might improve the learning process? <i>Quantitative measurements:</i> M1.1: Time taken to understand the pattern
Issue 2 (I2): Ease of pattern use	Q2.1: What problems or obstacles did you encounter when you used the pattern? Q2.2: What facilities did you find the proposed solution offered? Q2.3: If you were unable to use the solution in any activity, why? Q2.4: How can the proposal be improved to further ease of use? Q2.5: Was the solution modified or were additional artefacts generated as a result of pattern use? <i>Quantitative measurements</i> M2.1: Time taken to implement the pattern M2.2: How many elements within each activity were affected by solution use (impact level)?
Issue 3 (I3): Result of pattern application	Q3.1: Do the final applications include the usability functionalities? Q3.2: Does pattern use have any appreciable benefits? Q3.3: Would you use the patterns again in future development projects? <i>Quantitative measurements:</i> M3.1: Complexity level

As regards pattern understanding, the pattern learning curve is sizeable. At the level of process activities, the time taken by developers to understand the patterns at analysis and design level was found to be different to the time that they needed to understand the programming patterns. Whereas it took both developers a similar length of time to understand the patterns at the analysis and design level, there was found to be a significant difference (over 14 times) at the programming level. One possible explanation is the difference between the programming styles of the two developers and the additional activities they performed in order to learn the patterns. For example, one of the developers took much longer to understand the patterns because he took it upon himself to build demo applications including the functionality of each UM.

The developers provided input to improve the problem-solving artefacts and make them easier to understand. This should make it easier for other developers to understand the patterns. The end result was that developers acquired a full understanding of each pattern and generated improvements. This is consistent with the pattern concept. Note that demo applications are an essential part of programming patterns, although their inclusion does not guarantee learning or final application, and a priori knowledge of the versions of the technologies in which the code operates is important.

5.2. Ease of pattern use

As regards this second issue, although the developers consider that the solution is very complex, they had no trouble including the patterns in the analysis and design class diagrams. They did, however, find it hard to adopt the patterns in the sequence diagrams. Looking at Table 9, for example, it is clear that both developers found the impact level of the AO UM on application functionalities to be high. In fact, 87% and 82% of uses cases are affected by the AO UM functionality. On the other hand, as mentioned in Section 4.1, 22 scenarios were identified for the AO UM. The selection of the right scenario depends on the UM functionality required by each use

case. Therefore, more components have to be taken into account when the use case sequence diagrams are built. Although the impact for the PF UM is low (Table 9), the interaction between its components is, as shown in Fig. 6, complex. This complexity will mean that they are harder to include in the sequence diagrams of the affected use cases. The developers concluded that this issue could be improved by hiding the details of the pattern behaviour behind a façade component.

As regards the design-level facilities, both developers agreed that it is easy to find interfaces that connect the UM design with the system design, communicated via a single connection point (façade). They drew attention to the fact that the UM designs clearly show method calls from any system object.

The programming-level problems were related to the incompatibility between the technologies used by the developers and the programming pattern technologies, like the jQuery library and the Java Server Faces framework used by one of the developers. The other developer was unable to use the widget proposed for the PF UM with the VB .Net controls. To solve the technology incompatibility problems, some adaptations were made before implementation went ahead. One was a small change to the Javascript code. This provided for the straightforward implementation of the AO UM and also generated another version of the library compatible with other Java technology. The VB .NET developer replaced the ASP.NET by html controls and was able to use the PF UM widget without progress information.

Both developers underscored that demos were important both for understanding the design and code and for confirming that the code works. Another issue on which the developers agreed is that it is helpful for the programming patterns to be presented as tutorials, using a structured format describing the specific problem-solving steps and their implementation in pre-coded libraries. Finally, they also agreed that the AO UM pattern is easy to use second time round.

The developers also measured the number of new classes added by each UM. Table 10 shows the percentage increase of system classes when using the patterns. We found that although the percentages vary, the ratio is the same. This is only logical because the design-level solution is the same even though the code varies depending on the language used.

Table 9. Percentage of affected use cases (UC) in each case study (CS)

Usability mechanism	No. affected UC/ Total No. UC in CS1	% CS1	No. affected UC / Total No. UC in CS2	% CS2
<i>Abort Operation</i>	13/15	High (87%)	18/22	High (82%)
<i>Progress Feedback</i>	4/15	Low (27%)	7/22	Low (32%)

Table 10. Number of affected classes

Usability mechanism	No. new classes / Total no. classes in CS1	% CS1	No. new classes / Total no. classes in CS2	% CS2
<i>Abort Operation</i>	3/34	9%	3/18	14%
<i>Progress Feedback</i>	7/34	21%	5/18	22%

5.3 Result of pattern application

With respect to the third issue, developers evaluated several issues in order to establish the complexity level. Complexity can be measured by the number of classes used, number of messages exchanged by classes and internal logical complexity of the code of each class. The message exchange level is defined as the use of methods or attributes defined in one class by another. This is called the level of coupling, and, by definition, reuse will be harder if coupling is high. In this case, we make a distinction between the level of coupling between the UM functionality classes and the level of coupling between the UM functionality and the application classes.

For the AO UM, we found that the level of coupling between the pattern classes is high, but the logical complexity of each class is low. The complexity of the internal message exchange is hidden by using a façade class providing a single point of connection with the UM functionality. On the other hand, the level of coupling between the UM and applications functionality is high, ranging from at least two up to six calls per application scenario. This coupling level depends on the number of options available in each UM-related application. For example, the level of coupling is substantially greater if an information update form contains the quit application, cancel operation and clear form options.

For the PF UM, we found that both the level of coupling between the UM classes and the complexity of the internal logic of the classes is high. The level of coupling with the applications functionality is lower than for the AO UM and is mostly no greater than one on the client side, although it can be as much as two at server-side design level. The UM functionality is again used through a façade class that encapsulates the UM functionality.

As a result of the evaluation of the complexity level, the developers concluded that, even though the AO UM is highly coupled with the applications functionality, reuse is feasible because it calls and uses the same methods over again. So, it is really just a matter of copying and pasting to the parts of the application where it is required. Additionally, because the complexity of the internal logic of the UM is low, it is easy to understand how the code works and, if necessary, make changes.

In the PF UM case, on the other hand, although a single point of connection is used on the client side, independent processes also have to be implemented on the server side. These processes are executed in parallel to the main application process (multithreading) in order to gather progress information. This leads to a design change and an increase in code complexity. Additionally, if, as happened to developer 1, the technologies are incompatible and the internal code of the UM does not work properly, its complexity will prevent possible alternative solutions from being easily understood and found.

Another case identified by developers is where the AO UM and the PF UM are both implemented for one and the same application functionality. In these cases, there is a lot more interaction, but the required functionalities can be implemented because each UM operates independently.

6. Discussion

The proposed solutions are based on previous research that has identified the usability recommendations associated with the AO UM and the PF UM. The recommendations are stated as specific functionalities to be built into systems. These functionalities are taken into account from the start of the development process using requirements elicitation guidelines. This assures that the required functionalities will be implemented in the applications. The solutions have been used successfully on at least three occasions, thereby satisfying one of the premises of the pattern concept. Also both the original developments and the case studies used for evaluation are based on real requirements. Even so, the solutions are open to continuous improvement and will need to be applied in many other developments.

6.1. Threats to proposal

The results of the evaluation cannot be generalized as only two case studies have been conducted. However, it is noteworthy that the final results are successful and useful. They are successful in the sense that developers were able to understand and use the reusable elements with which they were provided. In the case of the AO UM, they also achieved the expected results, that is, the final

applications provided the usability functionality. They are useful in the sense that, where they failed to achieve the expected outcome, the causes were identified and the solution could be improved.

The level of abstraction is another threat to the solution. At a very low level of abstraction like programming, there will be similarly effective alternative ways of implementing the same functionalities. On this ground, the proposed implementation-oriented design patterns describe the implementation in a detailed but programming language-independent manner. Accordingly, expert programmers can rely on experience to perform the implementation. There are also limitations related to the application type. This study focused on web applications, whose features and behaviour differ from other application types. Usability functionalities can sometimes be harder to implement in web applications than in desktop applications.

We now discuss the different features of the proposed reusable solutions in order to grasp the implications of building each usability functionality into a web application.

6.2. Features of reusable solutions

As regards the way in which the UM components interact with the application components, we find that the two UMs (AO and PF) are highly coupled with application functionalities. This is illustrated by the sequence diagrams (Fig. 3 and Fig. 6) showing that the interaction between the UM components and the system components is high. Fig. 3 shows the sequence diagram for one of the AO UM application scenarios (highlighted in Fig. 2). This is the scenario where a user cancels an operation, the UM component recognizes that there are unsaved changes and asks the user if they should be saved, the user saves the changes, and another component associated with the UM executes the respective operation successfully. Taking into account that a total of 22 application scenarios were identified for the AO UM, there is a significant number of possible interactions between the UM functionality and the application.

However, one of the findings of the evaluation of pattern use is that this feature behaves differently for each UM at implementation level in web applications. In the AO UM case, communication with the application is confined to simple method calls that execute the odd functionality, the application functionality is unaffected because the AO UM components encapsulate everything that they need to know to execute their functionalities. Despite the fact that interaction with the application is high, it is performed via a single façade. Using a single façade component to represent the solution in the sequence diagram would simplify its representation, making the diagram and its respective programming more legible.

In the case of the PF UM, the application does have to perform additional functionalities in order to deliver information to UM components. Besides, either the application has to implement processes in separate threads or significant changes have to be made to the application in order to divide server-side tasks into smaller processes so that their progress can be checked from the client side. Both options increase development costs.

The AO UM is implemented using simple methods, and developers have no trouble studying and understanding their operation. On the other hand, methods associated with components for the PF UM are very complex, which is an obstacle to the developer being able to make changes. On top of this, the developer was in this case inexperienced in Javascript, and particularly the jQuery library, the technology used to develop the client side.

The internal complexity of the PF UM functionality is reflected in the sequence diagrams of its 12 application scenarios. For example, Fig. 6 shows the sequence diagram of one such scenario, called MultithreadedPIw/Infow/oCancelw/MSG, which is highlighted in Fig. 5. The sequence diagram shows that there are two cycles. One of the cycles is associated with the usability functionality and

serves the purpose of querying the progress of a process at set time intervals while the process is running. The other cycle is on the server side. It is associated with the application functionality and serves the purpose of periodically updating the active process progress information for query and display.

Analysing the above features of the UMs we find that implementing the HCI recommendations associated with each UM involves adding new functionalities with different coupling levels and complexity to the web applications. Based on a detailed study of the functionalities we believe that it is possible to establish intermediate solutions for implementing the usability recommendations. These solutions would have different levels of complexity at design and technological level and consequently different costs at implementation level. Some examples of these solutions follow.

For the AO UM, in response to the recommendation stating that it should be possible to cancel a process that lasts more than 10 seconds, two possible designs can be implemented: update the system state to the point where the process is cancelled, or implement a transactional design whereby, if the process is cancelled, the system reverts to its initial state without making any changes. For the recommendation stating that users should be warned if there are unsaved changes and asked whether they should be saved, there are also several options. One is for the system simply to warn users that there are unsaved changes and offer users the choice of going back and saving the changes. The other is for the system to warn users and know how to save the changes if users select this option.

For the PF UM, all of the progress indicators that need to provide progress information require additional system functionalities. They depend on the application functionality and the type of progress information that it can supply. Consequently, they will need to handle multithreaded processes. Other options that do not require the implementation of multithreaded processes are to subdivide a process into several tasks, whereby users can be informed of the number of completed tasks, or to simply use an indeterminate progress indicator until task completion.

As shown in Fig. 7, the functionality of each UM is represented by application scenarios, the functionalities are covered by responsibilities that are implemented by means of components. Components are converted into classes, whose interaction makes up the generic UM design. For example, in the case of the scenario highlighted in Fig. 2, which represents the case where a button on a form is pressed to cancel an operation, there are changes to be saved, the changes are saved successfully and the application reverts to the previous state. This scenario includes all the responsibilities defined for the AO UM. The system has to recognize that the cancel button has been pressed. If it does, the application must see whether there are changes to the form to be saved. If there are, a message must be displayed asking users whether they want to save the changes. If the user assents, the system must know how to save the changes and must save the changes. Finally, the system must know how to revert to the previous state. This illustrates how the responsibilities are inferred from the scenario, as shown on the left of Figure 7. The responsibility of listening for user actions and asking whether to save changes (R1) is associated with the UndoCancelFUF (Component 1). This is the same component as has been defined as a class that also operates as a façade for the functionalities of the AO UM (see Table 4 under the structure heading).

Implementing new solutions would mean analysing each and every one of these five levels in order to evaluate whether the new solution generates a new scenario and/or new responsibility. If a new responsibility is discovered, it has to be determined whether any of the existing components can fulfil the responsibility without loss of cohesion or whether, contrariwise, a new component has to be created. Depending on this, classes would either be modified or created. In the latter case, the interaction with the generic design would be established and the implementation-oriented design pattern modified. Finally, the code would be modified according to the final design, and all the programming patterns would be modified.

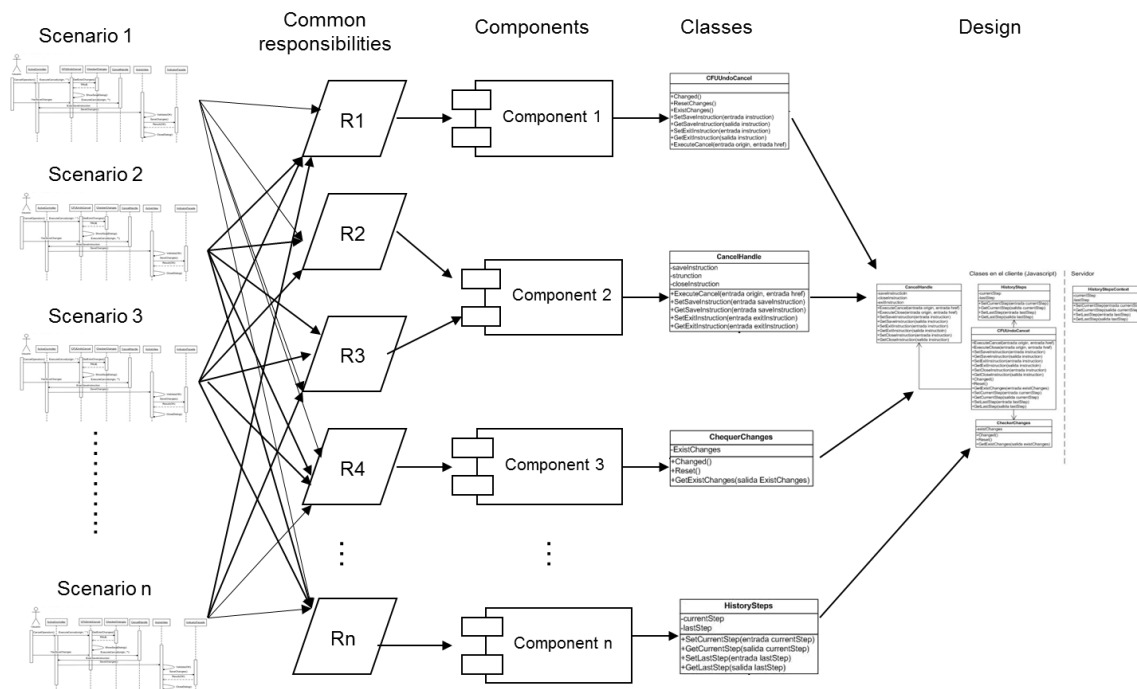


Fig. 7. Scenarios, responsibilities and components

7. Conclusions and future work

We have established that it is possible to output reusable solutions for later activities in the web applications development process. As the solutions target later activities in the development process, their level of abstraction is lower and they could be specified as implementation-oriented design patterns and programming patterns.

On account of the features of web applications, we get two types of reusable results for implementing the AO and PF UMs. One is for the client side and the other is for the server side. For the client side, we get a single design and code for each UM. The code is wrapped up as a totally reusable library. This is tested code that will enable the UM functionality to be efficiently implemented. It encapsulates the functionality of all the affected application scenarios and can be easily adapted to developments using other server-side languages.

As regards the server side, the implementation-oriented design is able to adapt the code for different applications even if they use different programming languages. The delivered code serves as an example, and there is a possibility of using the cut-and-paste option for other implementations.

The evaluation by independent developers pinpointed flaws in the documentation and the need to provide additional artefacts, like demo applications using real-world examples, to make the solution easier to understand. We also found that many of the reusable artefacts provided were useful. Although they took longer to understand and use first time round, the independent developers considered that they are potentially reusable in other implementations. As the definition of pattern implies, this solution is open to continuous improvement. With each new implementation, improvements will be able to be made, new functionalities added, the design refined and new useful code developed for other languages or versions.

As regards the PF UM, we found that even more functionality needed to be encapsulated for ease of use, because its internal complexity, plus the design and technological requirements to be met by

the web application, makes the solution hard to use. The developers agreed that the design of the PF UM was a good basis for finding a solution that meets the associated requirements and is easier to reuse. Despite these difficulties, the developer who used the VB .NET language was able to implement the indeterminate progress indicator using the respective pattern.

Both independent developers managed to use the AO UM design pattern and the Java and VB .NET programming patterns successfully. They managed to implement the UM functionality in the web applications. Even though the proposed solution for this UM is highly coupled with the application functionality, giving the impression of it being very hard to implement, developers found it very useful and think that they would use it in future developments.

Apart from the patterns, developers were also supplied with other artefacts used in the early activities of the development process. This means that effective pattern use depends on their being part of a larger solution. We conclude that it is useful but not sufficient to specify the reusable solutions identified as patterns. This is not surprising because the presented patterns target later activities within the development process and therefore depend on artefacts from the early process activities. The research process was able to establish the other artefacts upon which they depend: requirements elicitation guidelines and application scenarios with the respective sequence diagrams which provide an understanding of UM interaction with the application. Additionally, as they are programming patterns, they also include code libraries.

Some of the reusable solutions have been used successfully to build the usability functionality into the final application. As they have only been applied in the original three case studies and the two evaluations means, however, they have need of further use and improvement in other developments before the results can be generalized. On this ground, one of the future lines of research is to set up a web site to publish these patterns

As part of another line of future research we aim to look for and specify reusable elements as patterns for the Preferences UM and evaluate the results. The Preferences UM meets the criteria used to select the first UMs: impact level on design in terms of number of affected functionalities, ease of recognition by system users and ease of evaluation from the viewpoint of HCI guidelines. We will weigh up the possibility of enacting the same process for other UMs within the same family of FUFs in order to establish whether it is possible to share reusable elements across similar usability functionalities. Yet another line of research would focus on defining different usability levels and how they relate to more or less costly solutions at the level of web applications development.

Acknowledgements. This research has been funded by the Spanish Ministry of Science and Innovation “Tecnologías para la Replicación y Síntesis de Experimentos en IS” (TIN2011-23216) and “Go Lite” (TIN2011-24139) projects.

References

- Alexander, C., 1979. *The Timeless Way of Building*. Oxford University Press, New York.
- Alexander, C., Ishikawa, S., Silverstein, M., 1997. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.
- Alur, D., Crupi, J., Malsk, D., 2003. *Core J2EE Patterns*. Sun Microsystems Press Publisher, San Antonio Road, Palo Alto, CA.
- Bass, L., John, B., 2001. Supporting Usability through Software Architecture. *Computer*, 34(10), 113-115.
- Bass, L., John, B., 2003. Linking Usability to Software Architecture patterns through General Scenarios. *Journal of Systems and Software*, 66(3), 187-197.
- Bass, L., John, B., Kates, J., 2001. *Achieving Usability through Software Architecture*. Technical Report CMU/SEI-2001-TR-005, Software Engineering Institute, Carnegie Mellon University.

- Bass, L., John, B., Juristo, N., Sanchez-Segura, M.-I., 2004. Usability-Supporting Architectural Patterns, in Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society (ICSE 2004). Washington, DC, USA, pp 716-717.
- Biel, B., Seitz, S., Gruhn, V., 2011. Towards Pattern-based Generic Interaction Scenarios. *ScienceDirect*, 5, 771-775.
- Bosh, J., Juristo, N., 2003. Designing Software Architectures for Usability, in Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society (ICSE 2003). Portland, Oregon, USA.
- Brighton, 1999. The Brighton Usability Pattern Collection, <http://www.cmis.brighton.ac.uk/research/patterns/home.html> (Accessed June 2014).
- Bushmann, F., Meunier, R., Rohnert, h, Sommerlad, P., Stal, M., 1996. Pattern - Oriented Software Architecture. A System of Patterns. John Wiley & Sons, West Susses, England.
- Carvajal, L, Moreno, A.M., Sanchez-Segura, M.I., Seffah, A. Usability through Software Design. *IEEE Transactions on Software Engineering*, vol. 39(11), pp. 1582-1596, 2013.
- Folmer, E., Gulp, J., Bosch, J., 2003. A Framework for Capturing the Relationship between Usability and Software Architecture. *Software Process: Improvement and Practice*, 1, 67-87.
- Folmer, E., Welie, M., Bosh, J., 2006. Bridging Patterns: An Approach to Bridge Gaps between SE and HCI. *Information and Software Technology*, 48, 69-89.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1997. Design Patterns: Elements of Reusable Object-Oriented Software. 13 ed., Addison-Wesley, Reading, Massachusetts.
- Goodliffe, P., 2006. Code Craft: The Practice of Writing Excellent Code. No Starch Press, San Francisco, CA.
- Infragistics, 2015. Quince: Interactive user experience (UX) design patterns library, <http://quince.infragistics.com/> (Accessed Feb 2015).
- ISO, 2010. Ergonomics of human-system interaction. Part 210: Human-centred design for interactive systems, ISO 9241-210.
- John, B., Bass, L., Golden, E., Stoll, P., 2009. A Responsibility-Based Pattern Language for Usability-Supporting Architectural Patterns, in: Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems (ICSE 2004). New York, USA, pp. 3-12.
- John, B., Bass, L., Sanchez-Segura, M.-I., Adams, R., 2004. Bringing Usability Concerns to the Design of Software Architecture, in: Engineering Human Computer Interaction and Interactive Systems, vol. 3425 of Lecture Notes in Computer Science. Hamburg, Germany, pp. 1-19.
- Juristo, N., Moreno, A. M., Sanchez-Segura, M.-I., 2007a. Analysing the Impact of Usability on Software Design. *Journal of System and Software*, 80(9), 1506-1516.
- Juristo, N., Moreno, A. M., Sanchez-Segura, M.-I., 2007b. Guidelines for Eliciting Usability Functionalities Software Engineering. *IEEE Transactions on Software Engineering*, 33(11), 744-758.
- Laakso, S., 2003. User Interface Design Patterns, <http://www.cs.helsinki.fi/u/salaakso/patterns/> (Accessed June 2014).
- Mellarkod, V., Appan, R., Jones, D., Sherif, K., 2007. A Multi-Level Analysis of Factors Affecting Software Developers' Intention to Reuse Software Assets: An Empirical Investigation. *Information & Management*, 44, 613-625.
- Panach, J.I., Juristo, N., Valverde, F., Pastor, O. A Framework to Identify Primitives that Represent Usability within Model-Driven Development Methods. *Information and Software Technology*, In Press, 2014.
- Pattern Factory Oy, 2014. Patternry, <http://patternry.com/> (Accessed January 2014).
- Perzel, K., Kane, D., 1999. Usability Patterns for Applications on the World Wide Web, in: Proceedings of the Pattern Languages of Programming Conference (PloP '99).Monticello, Illinois, USA, pp. 1-17.
- Postmus, D., Meijler, T., 2008. Aligning the Economic Modeling of Software Reuse with Reuse Practices. *Information and Software Technology*, 50, 753-762.
- Runeson, P., Höst, M., 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Journal Empirical Software Engineering*, 14(2), 131-164.

- STATUS Project, 2001. Software Architecture that supports Usability, <http://www.grise.upm.es/rearviewmirror/projects/status/index.html> (Accessed June 2014)
- Szyperski, C., 2002. Component Software: Beyond Object-Oriented Programming. 2nd ed., Addison-Wesley, Boston, MA, USA.
- Tidwell, J., 2010. Designing Interfaces: Patterns for Effective Interaction Design. 2nd ed., O'Reilly Media, Sebastopol, CA, USA.
- Toxboe, A., 2015. User Interface Design Patterns Library, <http://ui-patterns.com/patterns> (Accessed Feb 2015).
- Van Duyne, D. K., Landay, J., Hong, J., 2006. The Design of Sites: Patterns for Creating Winning Web Sites. 2nd ed., Prentice Hall, Upper Saddle River, NJ, USA.
- W3C - World Wide Web Consortium. Web Design and Applications, <http://www.w3.org/standards/webdesign/> (Accessed June 2014).
- Welie, M., Trætteberg, H., 2000. Interaction Patterns in User Interfaces, in Proceedings of Seventh Pattern Languages of Programs Conference (PLoP 2000). Monticello, Illinois, USA, pp. 1-26.
- Yahoo, 2013. Design Pattern Library, <https://developer.yahoo.com/ypatterns/> (Accessed April 2014).