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:

International Journal of Software Engineering and Knowledge Engineering 25.4 (2015): 727 – 756

DOI: http://dx.doi.org/10.1142/S0218194015500084

Copyright: © 2015 World Scientific Publishing

El acceso a la versión del editor puede requerir la suscripción del recurso
Access to the published version may require subscription

International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

REUSABLE SOLUTIONS FOR IMPLEMENTING USABILITY FUNCTIONALITIES

Francy D. Rodríguez

Escuela 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. Acuña

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

Natalia Juristo

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

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Usability is a software systems quality attribute. Although software engineers originally considered usability to be related exclusively to the user interface, it was later found to affect the core functionality of software applications. As of then, proposals for addressing usability at different stages of the software development cycle were researched. The objective of this paper is to present three reusable solutions at detailed design and programming level in order to effectively implement the Abort Operation, Progress Feedback and Preferences usability functionalities in web applications. To do this, an inductive research method was applied. We developed three web applications including the above usability functionalities as case studies. We looked for commonalities across the implementations in order to induce a general solution. The elements common to all three developed applications include: application scenarios, functionalities, responsibilities, classes, methods, attributes and code snippets. The findings were specified as an implementation-oriented design pattern and as programming patterns in three languages. Additional case studies were conducted in order to validate the proposed solution. The independent developers used the patterns to implement different applications for each case study. As a result, we found that solutions specified as patterns can be reused to develop web applications.

Keywords: Software Engineering; Programming Patterns; Design Patterns; Usability.

1. Introduction

Usability is a critical software quality attribute critical in highly interactive systems [1]. Usability contemplates that specified users can use a product effectively and efficiently enough to achieve specified objectives in a specified context [2]. Apart from improving quality, several studies have pointed out other benefits of usability in software

development [3] [4] [5] [6] [7]: improved productivity of the work team and users and increased income from software projects.

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. The adoption of usability guidelines in software engineering (SE) has passed through several stages. At first it was considered sufficient to include usability features in user interface (UI) design to achieve satisfactory usability. As a result, usability was addressed in the later stages of the software development cycle. Software architectures tailored to this approach separated the UI functionality from the core functionality of the applications [8]. Model-view-controller (MVC) is an example of this type of solution.

Later it was found that some usability issues generate static and dynamic constraints on software components [9], that the separation strategy is no good for achieving a usable system and that some usability issues should be addressed from the early phases of the development cycle and particularly by the system architecture [10]. It was established that there is a relationship between usability and functional requirements and even that some usability-enhancing features have a direct impact on software functionality [11].

Based on HCI recommendations on how to improve software systems usability, Juristo et al. [11] identified three categories of guidelines depending on their effect on software development: usability guidelines with an impact on the UI, usability guidelines with an impact on the development process and usability guidelines 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. In turn, each HCI author identifies different FUF subtypes. Each subtype has been referred to as usability mechanism (UM) and has a name indicating its functionality. A non-exhaustive list of FUFs and their respective mechanisms is presented in [12].

In this paper we present reusable solutions for building three of the UMs identified as having a high impact on design into web applications. We selected three UMs: Abort Operation (part of the Undo/Cancel FUF), Progress Feedback (part of the Feedback FUF), and Preferences (part of the User Profile FUF). The other mechanisms belonging to these three FUFs are Global Undo, Go Back and Object-Specific Undo (Undo/Cancel FUF); System Status, Warning and Interaction (Feedback FUF), and Favourites and Personal Object Space (User Profile FUF). The UMs were selected according to several criteria: number of affected functionalities determined according to the features of the applications to be developed; ease of recognition by a system user, and ease of evaluation from the viewpoint of HCI guidelines.

The solutions that we propose aim to provide developers with tools for effectively building error-free usability functionality into a web application at the least possible cost. As the solutions that we present were discovered by implementing usability functionalities in different applications and successfully tested as part of other case studies, they have been specified as patterns. A pattern is considered to be a three-part rule that expresses a relationship between a given context, a problem and a solution [14].

Patterns are a way of specifying widely accepted reusable solutions within different branches of knowledge. They are useful for transmitting good practices in a standard format and language. A pattern is an experience-based reusable artefact, described in a structured format, which communicates designs and best practices [15] [16].

The reusable solutions for each usability functionality that we present are composed of several artefacts: a description of the functionality of the solution as application scenarios or functional requirements, a design, and code implementing the proposed design in three languages: PHP 5, Visual Basic .NET and Java. We refer to the union of design and code as programming pattern. Programming patterns, also known as idioms, are patterns with a low-level of abstraction. Programming patterns are a self-contained solution describing how to implement parts of or relationships between components identified in a design pattern using the programming language features and options [14].

This paper has been structured as follows. Section 2 analyses the related work dealing with usability patterns. Section 3 describes the applied research method, detailing the developed case studies. Section 4 analyses the process enacted to identify the application scenarios for the Abort Operation and Progress Feedback UMs and to describe the requirements associated with the Preferences UM. Section 5 details the process of specifying the solutions as programming patterns. Section 6 describes the evaluation of the proposed solutions based on another two case studies. Section 7 discusses the features of the solutions. Finally, Section 8 presents the conclusions.

2. Related Work

HCI researchers have defined a lot of usability-related patterns bearing different names: interaction or interaction design patterns [17] [18] [19], user interface patterns [20], usability patterns [21] [22], and web design patterns [23]. Although they are described or grouped differently, all these patterns have in common that they offer solutions to specific usability problems. There are also several pattern libraries for user interface design built by companies and available on the web [24] [25] [26] [27].

Some patterns appear in more than one definition or collection sometimes even under the same name. For example, the navigation aid pattern is consistently referred to as breadcrumbs in [18] [28] [24] [25] [26] [27], whereas the pattern indicating that the user requires a button or link providing the option of exiting a screen and returning to a familiar state is called escape hatch in [18] and home link in [28]. The HCI pattern definition explains what they are, and when and why they should be used, and provides detailed examples of what the UI should contain and how an application using the pattern should work. Most definitions do not detail how to design or implement the pattern at software development level. Only a few pattern library or collection web sites provide implementation examples for some patterns [28], specify code (html and css) for implementing the pattern [26] at UI level, and/or indicate which familiar language library controls are applicable.

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 solutions were developed that favoured the strategy of separating the UI from the core application functionality. 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) pattern and the presentation-abstraction-control (PAC) pattern. Later, however, the separation approach was found to be insufficient for implementing, debugging and maintaining some usability features [8].

Changes in the way that SE addresses usability have led to solutions covering the entire software development cycle being proposed and researched, that is, from requirements elicitation [12], through architecture [29] [8] [30] [10] [31] [32] and high-level design, to low-level design and implementation [33] [34]. A lot of research has focused on how to improve usability starting with the system architecture and identifies connections between usability features and software architecture [29]. Bass and John presented a set of usability scenarios in which the UI separation strategy is not good enough to produce a usable system and define architecture patterns to support usability [10]. John et al. [32] describe a study applying architecture patterns to support usability at business level. 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.

In the same vein, the STATUS project [31] examined the relationship between software architecture and usability and presented an approach for improving usability applying a specified design process. It proposes guidelines [12] for eliciting usability functionalities prior to architecture definition, useful for adding usability functionalities from the very first stage of the development process, namely requirements elicitation.

We find that hardly any of the above HCI and SE patterns provide details on low-level software design or implementation. In response, Folmer et al. [33] put forward the concept of bridging patterns. Bridging patterns are an extension of HCI patterns showing generic implementations for highlighting troublesome issues and their solutions. They include two more sections than HCI patterns: architectural implications and an example of the specific implementation in terms of classes and objects and/or in terms of technologies or techniques used. They intended to provide an instrument for improving communication on the boundary between SE and the HCI field. Folmer et al. describe four bridging patterns in [33]: Multi-level undo, Multi-Channel Access, Wizard and Single Sign-on.

We found that there are very comprehensive HCI pattern libraries and collections, but most do not provide details for implementing the software system. When they do, the implementation examples and code given are confined to usability features closely related to the UI and do not address usability functionalities that have been identified as having a high impact on design. Although they sometimes provide code for the odd feature, like Progress Feedback [26] [24], for example, classified as having a high impact on design, the suggested implementation is confined to the visual part of the usability functionality and does not deal with issues affecting the core application.

Our research follows Folmer et al.'s approach [33] in that it provides real implementations, but, unlike Folmer, we set out not only to clarify for architects the potential systems architecture and design implications of the usability functionality, but also to provide an implementation-level solution that can be reused as both a low-level design pattern and reusable code library.

Aspect-oriented programming (AOP) approaches implementation from a different angle [34]. AOP takes the view that there is a problem with using object-oriented design and programming: the interlinking of application functionality with usability functionality. Nevertheless, it remains to determine which usability features can be modelled as aspects and evaluate the real benefits of using this approach to implement usability functionalities.

3. Research Method

For this study we used a three-stage inductive research method, implementing case studies to induce a general solution. We started with three sets of real requirements for web applications and selected three UMs with a high impact on design: Abort Operation, Progress (or Long-Action) Feedback and Preferences.

The three case studies developed are 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.

The first stage of the research was to build the web systems assuring that all the final systems provided the functionality associated with the Abort Operation, Progress Feedback and Preferences UMs as well as their specific functionality. The elements related to the UMs (e.g. requirements, classes and code) were highlighted in each artefact generated in the development process, clearly identifying their respective UM functionalities as each application functionality may include more than one UM. For example, the functionality of each UM has a different typeface in the requirements documents; UM-related classes are coloured differently in the class diagram; UM-related components are shaded in the sequence diagrams (see Figures 2 and 4); and UM-related snippets are properly documented in the code.

The second stage identified the commonalities across the implementations of each UM in the three case studies and established which were reusable. The results were specified as patterns in the third stage of the research. We propose a single design for each UM and tailor the implementations to three programming languages: PHP 5, Visual Basic .NET and Java. The proposed design together with the implemented code is

specified as a programming pattern. As part of the third phase, we also extract common code snippets as a first step towards building a component library for usability functionalities.

4. Usability functionalities and multiple scenarios

The description of usability functionalities in the elicitation guidelines [12][13] is still too general for implementation. Developers using the existing guidelines face too many open options regarding the selected UMs (Abort Operation (see Web Appendix[a]), Progress Feedback (see Web Appendix[b]) and Preferences (see Web Appendix[c])). This may lead to scenarios in which the UM can be applied being omitted or to omissions or errors in their implementation. From an analysis of the requirements after including usability functionalities, we found that the functionality of each UM could be decomposed into more detailed application scenarios. Some scenarios had a major effect on design and implementation options or decisions.

We also found that the Abort Operation and Progress Feedback UMs differ substantially from the Preferences UM. Whereas the first two closely interact with the application functionalities, the Preferences UM behaves like any other system requirement and hardly interacts with the other functionalities at all. This means that in the first two cases usability functionality has to be described by means of scenarios representing system interactions, whereas the functionality of the Preferences UM is described by adding functional requirements to the system.

Trees with the identified combinations were built to give an overview of the scenarios discovered for the Abort Operation and Progress Feedback UMs. Each tree branch is a scenario. Each scenario has a name identifying its functionality and is described by sequence diagrams. We used two patterns recognized by the web developer community in order to generalize the sequence diagrams: façade pattern and model-view-controller pattern. The façade is used as an entry point to all usability functionalities. The view refers to the user interface, the controller receives the user events and sends requests to the respective components, and the model manages the business rules. In the following sections we detail the scenarios, requirements, responsibilities and components defined for each usability functionality.

4.1. *Scenarios for the Abort Operation UM*

The Abort Operation UM should enable the user to cancel an operation, a command or exit the application in a safe and predictable manner. The elicitation guideline for the Abort Operation UM divides the questions into three levels: application, operation and command. At application level, the guideline indicates that users should be asked whether an option for exiting the application is necessary and, if so, how the option should be

[a] http://www.grise.upm.es/sites/extras/7/Usability_Elicitation_Pattern_AO.pdf
[b] http://www.grise.upm.es/sites/extras/7/Usability_Elicitation_Pattern_PF.pdf
[c] http://www.grise.upm.es/sites/extras/7/Usability_Elicitation_Pattern_PREF.pdf

displayed to the user. According to the HCI recommendation associated with the elicitation question, the quit option 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 presented.

The operation level refers to actions requiring the execution of one or more steps within an application, each of which requires user interaction. 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 given to the application by means of a single user interaction, that is, pressing a button, clicking on a link, selecting a menu item or any other available option. The Abort Operation UM was defined as an alternative path to core functionality throughout the requirements description. Sequence diagrams were used to describe each alternative path associated with the UM.

We found that there is a relationship between the elicitation guideline questions, possible application scenarios, web system interpretations and the final system state. At application level, for example, if the response to the guideline question *Will the user need an exit option for the application*? is yes, there are two possible scenarios taking into account the HCI recommendation stating that there should be an option for saving changes: there are or there are no changes to be saved when the quit option is selected. If there are no changes, the system will go to the next state, which may be a login page. If, on the other hand, there are changes to be saved, the user should be asked whether or not to save the changes. If the user does not want to save the changes, the system will go to the next state, but if the user wants to save the changes, there are several possible scenarios: they are saved successfully, or validation errors or database errors occur while saving the changes. The application should go to a predictable state whatever the case.

We built scenario trees for each HCI recommendation level: application, operation and command. At the application level, we identified five scenarios. At operation level, however, the number of scenarios grew to 16, mainly because, unlike the application level which has only one possible source (exit option), there are four possible sources at the operation level: dialogue box containing a Cancel button, form containing a Cancel button, selection of another application option and Clear button.

Figure 1 shows the size of the scenario tree at operation level. Note that, despite the fact that the illustrated scenario tree contains 24 branches or cases, there are only 16 scenarios. This is because some cases generate one and the same scenario. In Figure 1, * is placed beside the scenario name to identify repeated scenarios. For example, the FormCancelOpButtonSaveChangesValError scenario is generated when the cancel option is executed from a form containing a button, there are changes to be saved, the user chooses to save the changes but validation errors occur while they are being saved. As far as UM functionality is concerned, the scenario for the cancel option that is generated when another application option is selected from a dialogue box or form, there are

changes to be saved, the user chooses to save the changes but validation errors occur while they are being saved is exactly the same.

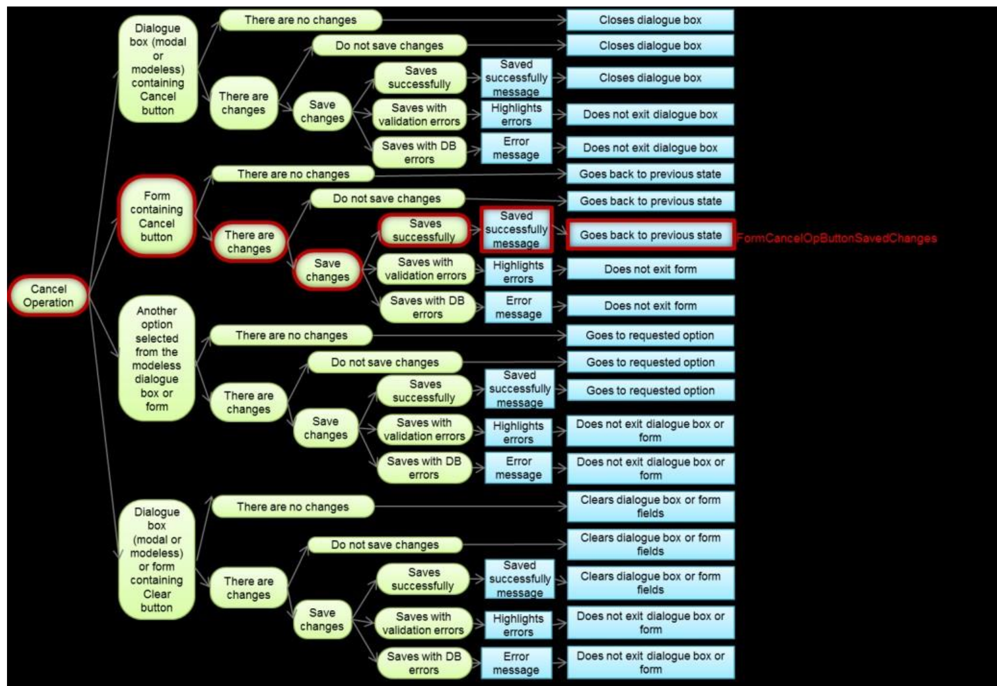

Fig. 1. Operational level scenarios tree for the Abort Operation UM.

Each branch of the tree shows a possible use case scenario for the UM functionality. Each branch was named and described using sequence diagrams, for example, the scenario for cancelling an operation using a form containing a cancel button, where there are changes, the changes are to be saved and are successfully saved was called FormCancelOpButtonSavedChanges (scenario highlighted in Figure 1). Figure 2 illustrates the associated sequence diagram.

The identified responsibilities for the Abort Operation UM are:

- Listen to 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.

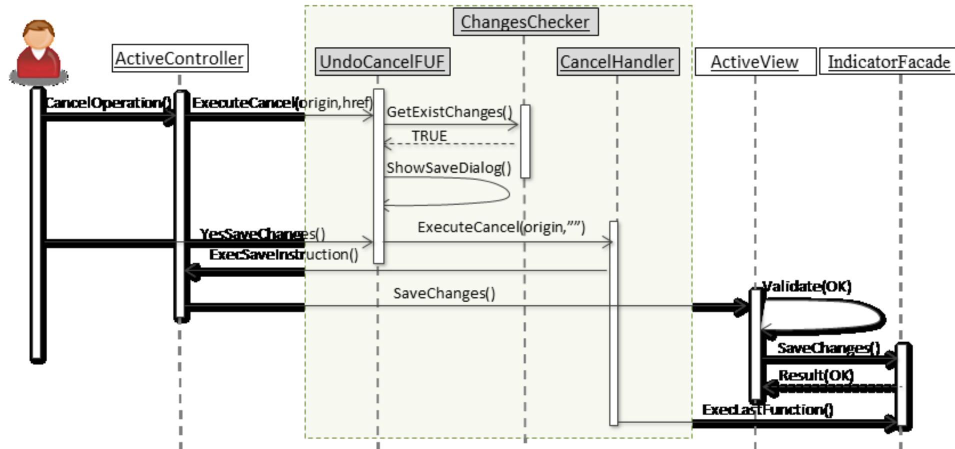

Fig. 2. Sequence diagram for the FormCancelOpButtonSavedChanges scenario.

We defined a set of components in order to fulfil the identified responsibilities. They are shown in Table 1. These components are used in the sequence diagrams. The three components related to usability functionality are shaded in Figure 2.

Table 1. Abort Operation UM component responsibilities.

Component	Responsibility
ChangesChecker	Updates and reports changes to be saved in the application
CancelHandler	Saves changes if operations are aborted and gets system into a state that is predictable and safe for users
UndoCancelFUF	Receives request 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	Updates and provides information on previous and current application states.

4.2. *Scenarios for the Progress Feedback UM*

The Progress Feedback UM informs the user either graphically or textually of process progress. The context for Progress Feedback functionality implementation is: 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?

The elicitation guideline also shows a summary of the HCI recommendations on which the questions are based. These recommendations provide more details about the issues to be taken into account in the implementation. For example, one HCI recommendation suggests that the system should supply information on the proportion of the operation that has already been completed and the time remaining to finish the task. For a web application, this recommendation implies major design and implementation decisions: either the use of asynchronous processes to discover the progress of an ongoing server-side task or the division of a process into several tasks for execution on the client side monitoring progress on a task-by-task basis. Table 2 shows a list of the elicitation guideline questions, their associated HCI recommendations, possible application scenarios and their technological implications.

We infer from the technological implications shown in Table 2 that there are conditions that generate significant design and implementation changes. These conditions include task criticality, the type of available progress information, process cancellability or usability functionality responsibility for reporting whether or not the process has finished.

Table 2. Relationship between elicitation questions and scenarios for the Progress Feedback UM

Questions/Recommendations	**Cases/Scenarios**	**Technological Implications**
Question: Which processes are critical? Recommendation: If the process is critical, users should not be allowed to do anything else until this task is completed. If the task is not critical and takes over 5 seconds, users should be allowed to run another operation if they so wish.	The process is critical. Users will not be allowed to do anything else.	A scenario that allows users to execute two simultaneous processes calls in web applications for the use of asynchronous processes and a checker to check events on all the navigable pages and monitor running processes. It will also require a server-side mechanism for reporting when a process has finished.
	The process is not critical and takes less than 5 seconds. Users will not be allowed to do anything else.	
	The process is not critical and takes longer than 5 seconds. Users will be allowed to run another operation if they wish.	
Question: How will the user be informed when the process has finished?	By displaying and automatically closing a message reporting the results (progress indicator will also be closed)	If the usability functionality is responsible for reporting that a process has finished, a mechanism should be implemented to query the process state.
	By displaying a message which will not be exited until it is closed by the user	
	By displaying and automatically closing a message on the progress indicator	
	By displaying a message on the progress indicator which will not be closed automatically	
	By displaying the actual result instead of a message	

Table 2 (Continued) Relationship between elicitation questions and scenarios for the Progress Feedback UM

Questions/Recommendations	Cases/Scenarios	Technological Implications
Question: How will the user be informed about the progress of each task? Recommendation: Regarding the remaining time: If the timing can be calculated, use either a Time-remaining progress indicator or a Proportion-completed progress indicator; if timing cannot be estimated, but the process has identifiable phases or tasks, use a Progress checklist; if neither of these possibilities exist, then indicate the number of units processed; if no quantities are known, use an Indeterminate progress indicator.	By displaying a time remaining progress feedback indicator	For processes requiring the provision of progress information that run as a single process on the server, asynchronous processes will have to be used to manage the core process and an independent process that queries and updates the progress on different threads. Besides, a mechanism should be added to the process to determine and report progress. Another possibility is to subdivide the process into n tasks, whose execution is monitored from the client in order to report progress in terms of the number of tasks. Another alternative would be a combination of both options.
	By displaying a proportion completed indicator	
	By displaying a progress checklist	
	By displaying a message reporting the number of processed units	
	By displaying an indeterminate time progress indicator	
	By displaying a time remaining progress indicator and number of processed units	
	By displaying a proportion completed indicator and the number of processed units	
	By displaying a progress checklist and the progress of one of the tasks on the list in terms of time, proportion or units	
Question: What information is necessary in each case? Recommendation: Show how much progress has been made either verbally or graphically and tell the user: what's currently going on, what proportion of the operation is complete so far, how much time remains.	By displaying the process name	The graphical component of the progress indicator must be able to display all four options identified by the scenarios and their possible combinations, and be able to be configured to display them according to the available information.
	By displaying the lower or upper value bounds, for example, 0% to 100%, task 1 to n, 0 to x registers, total time, etc.	
	By displaying the current progress value	
	By displaying a description of the phase or task as part of the overall process	
Recommendation: The indicator must tell the user how to stop (or cancel) the operation if the time remaining is longer than 10 seconds.	If the operation cannot be cancelled, the actions open to users depend exclusively on whether or not the task is critical.	If a task is cancellable, the functionality necessary for finishing the process leaving the system in a safe and predictable state should be implemented.
	If the operation can be cancelled, display a cancel option for users.	

Apart from the above implications, there is another variable to be taken into account: the technology. The design will be different depending on whether or not the technology is able to manage multiple threads of execution (single-threaded or multithreaded technology). A multithreaded language is one in which several processes can be executed simultaneously, each with their own control flow. If the technology is multithreaded, separate server-side processes can be used to update process progress. However, if the technology is single threaded, no other task will be able to be run simultaneously to query progress until the primary process finishes. There are two options in this case: use an indeterminate progress indicator or change the design in order to subdivide the process into several tasks that execute and display the processed tasks one by one, for example.

We found that there are 12 application scenarios for the progress feedback functionality. The whole tree is shown in Figure 3. The scenarios are conditioned by the possible responses to the elicitation questions and by the type of technology used. Note that the nodes nearest to the tree root are related to responses to elicitation questions, whereas the terminal nodes depend on the technology features.

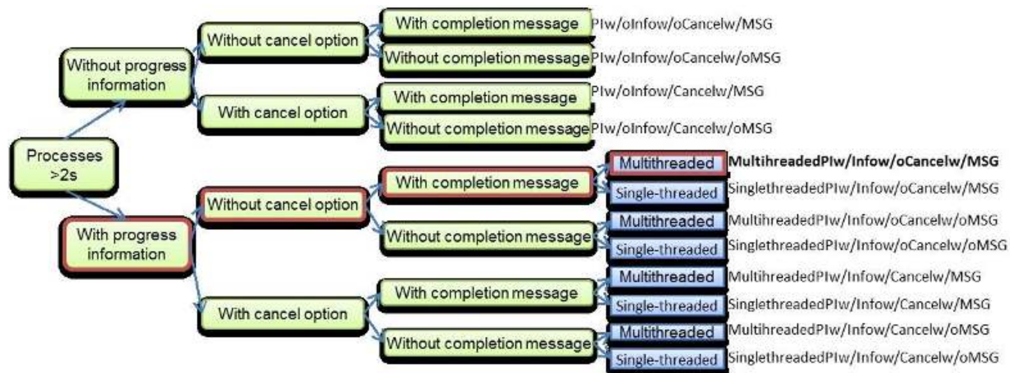

Fig. 3. Progress Feedback UM application scenarios

Each scenario is described by a sequence diagram. For example, we have denoted the scenario where a non-cancellable process using multithreaded technology with progress information has to display a completion message for the user (highlighted in Figure 3) as MultithreadedPIw/Infow/oCancelw/MSG. Figure 4 shows the respective sequence diagram. The sequence diagram shows that there are two cycles. One cycle 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.

The responsibilities identified for the Progress Feedback 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. They are described below.

ProgressFeedbackUI. This component displays the either 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 that the process has finished as instructed. It displays the Close or Cancel button and a completion message when necessary. It closes the progress indicator.

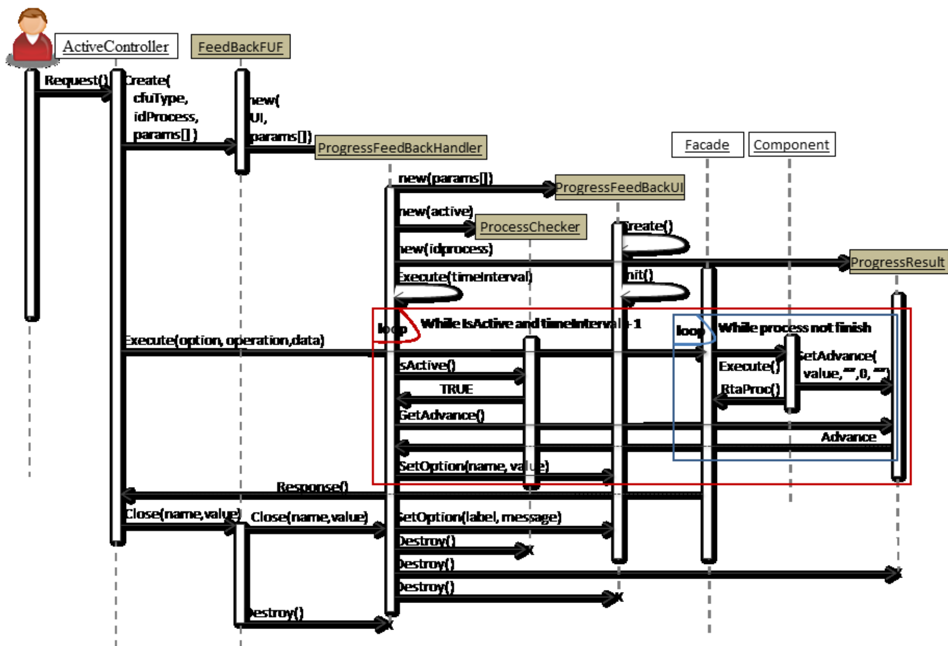

Fig. 4 Sequence diagram for MultithreadedPIw/Infow/oCancelw/MSG

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 session process progress information. 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.

4.3. *Preferences UM*

The Preferences UM does not interact much with the remainder of the system and behaves like an add-on functionality. It is addressed in the same way as a functional requirement by defining use cases, conceptual model and sequence diagrams. The Preferences UM allows users to define and save their own settings for aspects like language, font, icons, colour schemes and sound use. The user settings should be saved for subsequent sessions. There is also the option of selecting predefined preferences settings.

We identified a set of requirements associated with the Preferences UM to be taken into account:

- Preferences can be configured at user, user group and application level.
- There must be a basic preferences configuration. This will be the default configuration assigned when creating a user, user group or when the application starts up without a login process.
- Apart from the basic or default configuration, other pre-established preferences settings may be defined. These settings will be available for users to use to change their current settings or apply when creating a user group, user profile or user.
- A persistence mechanism is necessary to store preferences information.

We found that four functional requirements cover the Preferences UM functionality:

- Apply user preferences settings during the application login process.
- Change preferences.
- Change preferences for a predefined set of preferences.
- Change the application language.

Each requirement has a sequence diagram that describes its behaviour. Figure 5 shows the sequence diagram for applying preferences during the login process.

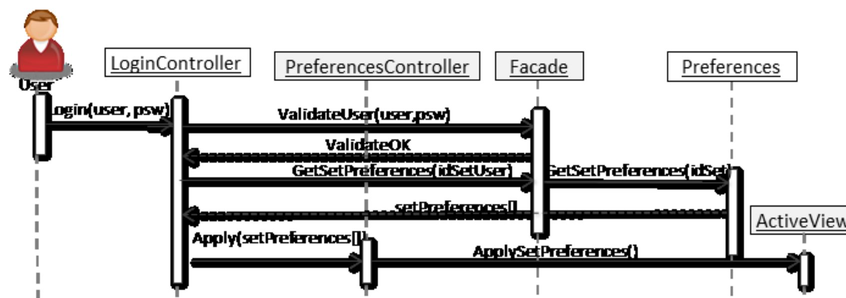

Fig. 5. Sequence diagram for “Apply preferences during login process"

We use the components described below in order to meet the four requirements associated with this usability functionality.

Preference_type. This component is the persistence mechanism for the available preferences settings types. It includes the form of physical storage and the methods for representing and accessing information.

Preferences. This component is the persistence mechanism for possible preferences settings. It includes the form of physical storage and the methods for representing and accessing information.

PreferencesFacade. This component represents the entry point from the web layer to the business components and the model data. It provides all the methods required by the controller to meet any user request. Internally it can call any business layer component and access, modify and query the data.

PreferencesCSS. This component is a class for dynamically generating style sheets. It is the key for changing the visualization of the pages when users change any of their preferences. It will not work without the id of the set of user preferences.

PreferencesController. It is the component that has the responsibilities associated with the Preferences UM at client level. It applies a set of preferences to the current view of the application or changes the application language. It also displays and is used to modify the current preferences settings. To fulfil these responsibilities, the component is based on a PreferencesCSS class that is able to dynamically generate the style file for each page and the functions of the façade.

5. Reusable solutions for the Abort Operation, Progress Feedback and Preferences UMs

After identifying scenarios, requirements, responsibilities and components, we continue to analyse the result of the three case studies searching for matches in classes, methods, attributes and code. Based on the findings we were able to define a single design and tailor the implementations to the three different programming languages: Visual Basic .NET, Java and PHP 5. The programming patterns that we propose cover all the discovered scenarios and requirements. We know that other implementations will apply subsets of these scenarios and requirements on which ground not all the implemented code will always be used. The example code associated with the patterns has been documented so developers can easily locate the useful parts depending on the scope of the UM functionality that they need to implement.

5.1. *Abort Operation UM*

After examining the elements related to the Abort Operation UM in the class design for all three case studies, we found that:

- All three designs have a class that operates as a façade. As a result, all three cases were built to be reusable solutions.
- In all three cases, there is a class that has the responsibility of determining whether there are changes to be saved. This class has only one attribute and three methods with similar functionality. The attribute is able to find out whether there are changes to be saved, and the three methods are able to modify and query the attribute value.
- In all three cases, there is a class that encapsulates the main methods in order to respond to a request to exit the application or cancel an operation or command. It

knows how to save changes and what the state of the system should be after completing the request.

- Another class appears in two out of the three case studies. This class is responsible for storing the information on the previous and current system state. VB .NET has no class for this purpose because it uses technology-specific features.

At attribute level, we found that although attribute definition is technology dependent, 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.

We also found that other design decisions matched. This applies to the use of a singleton class as a façade (UndoCancelFUF) because the solution requires several unique session attributes. The singleton pattern assures that there is only one instance of the class and consequently a single data update channel. In the case of the Abort Operation UM, the session data that should be unique are: whether or not there are changes to be saved, the latest application state, how the current changes should be saved and which dialogue box is active. The main difference that we found was how the system states were handled. Due mainly to the Java technology used (JavaServerFaces), a server-side class was used to store the information on the previous and current system state, whereas VB .NET and PHP 5 used session variables instead of classes.

From the analysis of the three designs, we concluded that many attributes, methods and classes fulfilled the same responsibilities and could therefore be unified in a single design shown in Figure 6. The elements that do differ are complementary rather than mutually exclusive and specify a design that covers all the discovered application scenarios.

At programming level, we found that a significant proportion of the logic is on the client side in all three cases. Being web systems, the client-level implementation is written in the same script language in all three cases: Javascript. Some changes were made to unify the three script codes, and a single piece of code was generated for all three systems covering all the identified scenarios. Although the components have the same responsibilities, the design is modified for adaptation to the technology.

Table 3 shows the unified design proposed for the Abort Operation UM, specified as a design 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 and, as this is an implementation-oriented design pattern, includes an implementation section that specifies the steps necessary to codify the proposed design.

Therefore, we have one design proposal and its implementations in each language. The web appendix shows the programming patterns for the Abort Operation UM in three

languages VB .NET[d], Java[e] and PHP 5[f]. The programming pattern template is not the same as the design pattern. The structure section shows the modified design tailored to the technology features. There is a new example section which shows the real code used to implement each step of the solution.

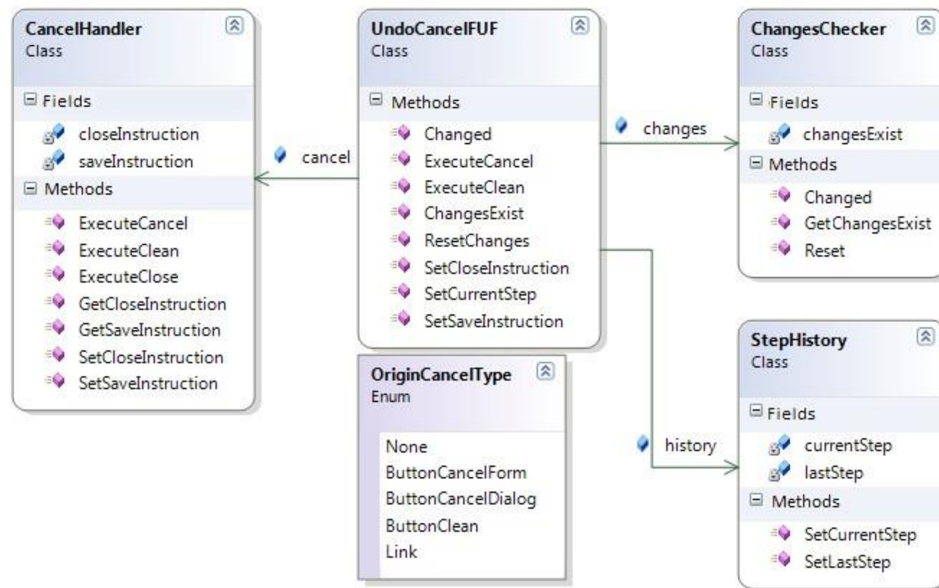

Fig. 6. Design of unified classes for the Abort Operation UM

5.2. *Progress feedback UM*

In all three case studies, the design of the Progress Feedback functionality manages the same five classes as described as components in Section 4.2. Figure 7 shows the final class diagram. Because we are dealing with web applications, a distinction has to be made between the client-side and server-side code. The client-side code is programmed in the same script language for all three cases: Javascript. The server-side code, on the other hand, is fully technology dependent, and therefore the proposed design varies in each case. At client level, we were able to generate a single piece of Javascript code which is reusable across web applications irrespective of the technology that they use. The proposed programming patterns for the three languages used are shown in the web appendix: Visual Basic .NET[g], Java[h] and PHP 5[i].

[d] http://www.grise.upm.es/sites/extras/7/PP_AO_VB_NET.pdf
[e] http://www.grise.upm.es/sites/extras/7/PP_AO_Java.pdf
[f] http://www.grise.upm.es/sites/extras/7/PP_AO_PHP.pdf
[g] http://www.grise.upm.es/sites/extras/7/PP_PF_VB_NET.pdf
[h] http://www.grise.upm.es/sites/extras/7/PP_PF_Java.pdf
[i] http://www.grise.upm.es/sites/extras/7/PP_PF_PHP.pdf

Table 3. 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
SOLUTION Components are required to fulfil the responsibilities associated with the UM. They are: • A component to update and report on whether there are any changes to be saved in the application. • A component that queries whether there are 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.	
STRUCTURE CancelHandler Class: Fields: closeInstruction, saveInstruction; Methods: ExecuteCancel, ExecuteClean, ExecuteClose, GetCloseInstruction, GetSaveInstruction, SetCloseInstruction, SetSaveInstruction UndoCancelFUF Class: Methods: Changed, ExecuteCancel, ExecuteClean, ExistChanges, ResetChanges, SetCloseInstruction, SetCurrentStep, SetSaveInstruction ChangesChecker Class: Fields: existChanges; Methods: Changed, GetExistChanges, Reset StepHistory Class: Fields: currentStep, lastStep; Methods: SetCurrentStep, SetLastStep OriginCancelType Enum: None, ButtonCancelForm, ButtonCancelDialog, ButtonClean, Link cancel, changes, history	
IMPLEMENTATION • Create a singleton UndoCancelFUF class. • Create a ChangesChecker that updates and provides information on application changes. • Create a StepHistory class that updates and provides information on the previous system state. • Create a CancelHandler class that knows how to save operations, clear fields and close dialogues and which the next system state is after an operation is aborted. • Implement the UndoCancelFUF class methods to operate as a façade for the CancelHandler, StepHistory and ChangesChecker classes. • 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. • Implement the right functionality so that the system always knows which method to use or which action to take to save a change after cancelling an operation or quitting an application. This can be done using the CancelHandler class. • 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. • 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. • Implement the right system functionality to save the latest state during application navigation so that this data item is available if a previous state has to be restored. This can be saved in the StepHistory class.	
RELATED PATTERNS	Singleton Pattern and Façade Pattern.

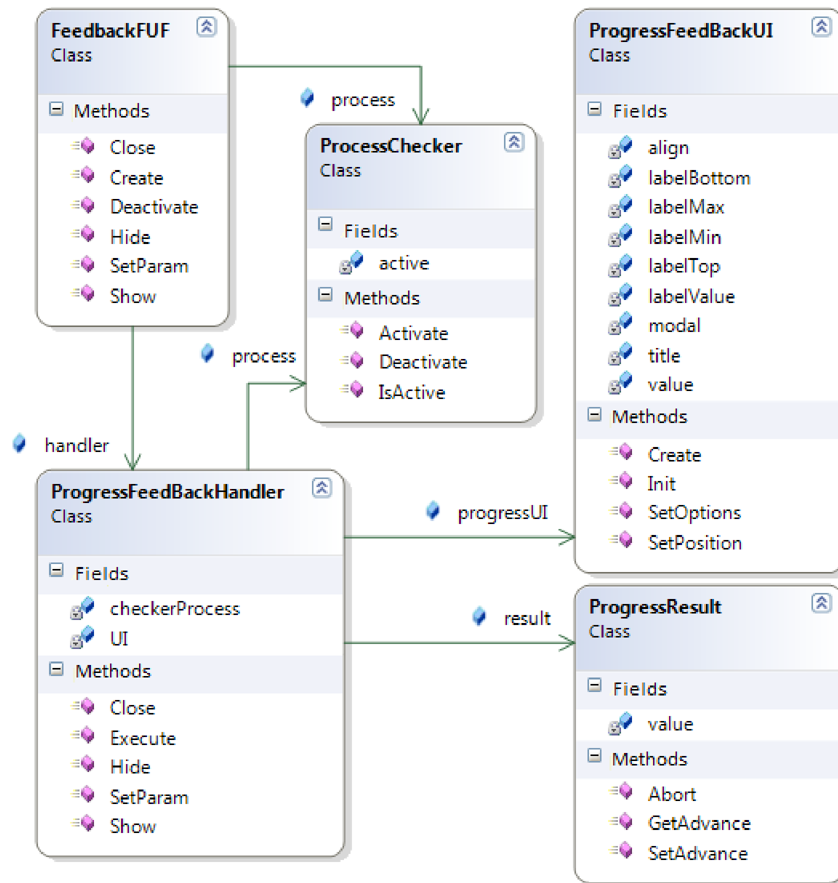

Fig. 7. Unified class design for the Progress Feedback UM

5.3. *Preferences UM*

The components identified for the UM were described in Section 4.3. As mentioned in Section 4.3, this usability function is not described using scenarios because it hardly interacts with application functionality at all. In this case, the UM functionality is specified as four functional requirements. Figure 8 illustrates the class diagram that covers the four established requirements. We find in this case that the usability functionality has components in all three web application layers: persistence, business and (web) interface.

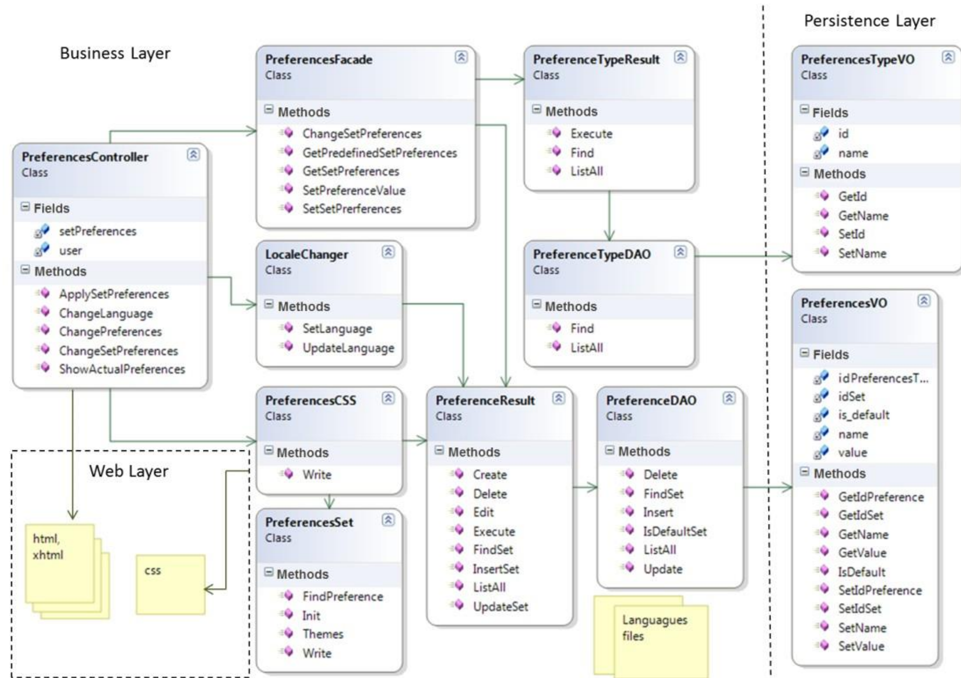

Fig.8. Class design for the Preferences UM

As for the Abort Operation and Progress Feedback UM patterns, we obtained unified client-side Javascript code for all three case studies. The web appendix shows the programming patterns for the Preferences UM in Visual Basic .Net[j], Java[k] and PHP 5[l].

6. Evaluation

We have used cases studies [35] as a research methodology in order to evaluate the feasibility of using design and programming patterns to implement usability functionalities in web applications. The case study methodology is suitable for use in software engineering research because it studies current phenomena in their natural setting. It is used when the boundary between the phenomenon and its setting is not very clear. By definition, case studies are conducted in real-world scenarios and are highly realistic in return for which they are less controllable. Case studies mostly use qualitative data that provide in-depth descriptions. However, quantitative data can be used too. Case studies do not provide statistically significant conclusions. On the contrary, they use different types of evidence, such as data, assertions and documents to support relevant conclusions.

[j] http://www.grise.upm.es/sites/extras/7/PP_PF_VB_NET.pdf
[k] http://www.grise.upm.es/sites/extras/7/PP_PREF_Java.pdf
[l] http://www.grise.upm.es/sites/extras/7/PP_PREF_PHP.pdf

The research process is similar to any other type of empirical study: case study design, preparation for data collection, data collection, data analysis and reporting. Case study design is flexible and the steps are quite often reiterated. Thanks to their flexible design, the primary study parameters can be changed in the course of the study. The only exception is the originally specified objectives, as this would alter the purpose of the case study. The data were collected mainly by means of questionnaires, semi-structured interviews and document analyses. Most of the analysis was carried out using qualitative methods combined with a limited quantitative analysis.

The proposed solution is evaluated for exploratory purposes. The aim is to discover what happens during the development of web applications when using design and programming patterns in order to implement three UMs: Abort Operation, Progress Feedback and Preferences. The problem context is highly interactive web applications developed using the object-oriented paradigm. The unit of analysis is the web application. The case study is embedded and uses two units of analysis: two web applications developed using the proposed patterns.

The web applications used as units of analysis have been built by separate developers with programming experience. The developers built the case studies as part of their Madrid Technical University master's theses. One of the developers holds a BS in Computer Science and Engineering, an 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.

They developed different applications based on real requirements. One of the cases 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, ordering and reception system. The developer was given a preliminary requirements document containing 13 functionalities. The second case 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. In order to elicit requirements, the developers used the same elicitation guidelines as were used in this study, plus reusable artefacts output by this research:

- Application scenarios for Abort Operation and Progress Feedback UMs and the requirements definition for the Preferences UM.
- Design patterns. They provide a description of the components required to fulfil the responsibilities associated with each usability mechanism.
- Programming patterns. They show a real-world solution using the design pattern with specific technology. They provide reusable code snippets.

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 able to ask anything they wished 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.

The developers were asked to document their reaction to the proposed solutions throughout the entire process and rate how useful each part of the solution was. In particular, they were asked to rate three aspects: ease of pattern understanding, ease of pattern use and result of pattern application. The developers also recorded information on time taken, number and type of elements affected by the solution. Finally, an interview was held to find out how the developers rated the process as a whole.

The final documents delivered by the developers are secondary and tertiary data sources. The secondary sources are the parts of the documents where the developers directly respond to the research questions and the tertiary sources are the parts of the document related to all the artefacts generated during the development process: requirements specification, design and code.

From the data analysis, we found that the developed web applications adopted two out of the three UMs: Abort Operation and Preferences. In the case of the Progress Feedback UM, the Java programming pattern was not applicable because the JQuery framework used in the pattern was incompatible with the JavaServerFaces technology used by the developer. However, the developer did think that it would be possible to use the same design if code were generated in the technology that he used. The Visual Basic.Net implementation was also troublesome, and only the Progress Feedback UM scenario reporting no progress information was implemented.

With regard to the quantitative data, the developers took some measurements of the impact of using the proposed patterns on their systems. One is the number of functionalities affected by each UM. As shown in Table 4**,** each UM has an equivalent percentage impact. In both case studies, the Abort Operation UM has a high impact on systems because it affects over 80% of the system use cases, whereas the Progress Feedback and Preferences UMs do not have much impact in terms of the number of affected use cases.

Table 4. Percentage of use cases (UC) affected 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
Abort Operation	13/15	High (87%)	18/22	High (82%)
Progress Feedback	4/15	Low (27%)	7/22	Low (32%)
Preferences	1/15	Low (6.7%)	1/22	Low (4.5%)

The developers also counted the number of new classes added by each UM. Table 5 shows the percentage increase of system classes when using the patterns. We found that although the percentages vary, the ratio is the same, that is, the Preferences UM has the least and the Progress Feedback UM the most impact in all three cases. This is only logical because the design-level solution is the same even though the code varies depending on the language used.

Table 5. Number of affected classes.

Usability mechanism	No. new classes / Total No. classes in CS1	% CS1	No. new classes / Total No. classes in CS2	% CS2
Abort Operation	3/34	9%	3/18	14%
Progress Feedback	7/34	21%	5/18	22%
Preferences	1/34	3%	2/18	10%

Another measure is the effort in terms of time taken to add UMs. These measurements are not comparable because they are very much influenced by the development model used, and each developer's experience and programming style. Some examples of these differences are: one of the developers chose to use UML, whereas the other decided to use a tool to automatically generate the models. One developer used paper prototypes, whereas the other built an operational prototype. One developer decided to build a demo to find out how the Abort Operation UM worked and the other decided to follow the pattern code. Despite these differences, there are some points in common: it took both developers what they considered to be a long time to understand each pattern at first, and both had to ask for further explanations on how the patterns worked.

The developers concluded that it takes quite a lot longer to use the patterns first time round because users have to find out how they work, but they can then be successfully used to implement the usability functionality. The design pattern was easier to use and was applied in 100% of the cases. As regards implementation, there were two possibilities on the client side: the code was either used as a black box or tailored slightly. Tailoring was necessary because of incompatibilities between the language technologies or versions. On the server side, the code could not be used as a black box. However, a copy/paste schema was feasible. Finally, developers concluded that once they grasped the purpose of the scenarios, they were useful and easy to use as a complement for elicitation guidelines and in the analysis stage.

In response to the questions asked to validate the proposal, developers concluded that although it takes longer to understand the solution first time round, the final result was positive, as the systems provided all the usability functionalities, except the Progress Feedback UM. However, they did think that this UM could be implemented using the proposed design pattern if it were reprogrammed for the technology used. They suggested several improvements for the artefacts, such as better documented code or demo application development, but they did say that they would use the provided solution in other developments.

7. Discussion

After implementing the Abort Operation and Progress Feedback usability mechanisms in three applications, we found that there were multiple application scenarios depending on user responses to the elicitation guideline questions. No such scenarios were identified, however, for the Preferences UM because this functionality interacts less with application functionality. We found that the more scenarios there are, the more coupled usability functionality is with application functionality. Application complexity is directly proportional to the number of scenarios.

The three case studies were comparable because they were developed according to an object-oriented approach despite being implemented in two languages that are not traditionally used with this approach: PHP and Javascript. Although these languages are not usually used in object-oriented programming, they are able to define classes and methods. This provides points of comparison.

We found that each UM had to fulfil similar responsibilities in all three case studies. This means that there are also similarities in the design and coding. So, there are components that fulfil the common responsibilities. This does not necessarily mean that there is one component for each responsibility. In some cases, one component is used for one responsibility and in others one component is used for two or three similar responsibilities.

Some components could be implemented without making any distinction regarding the scenarios that they were going to fulfil, that is, the code fulfils its associated responsibility without reference to scenarios. In other cases, see Figure 9, distinctions had to be made according to some conditions specified by the scenarios. Parts of the code will not be executed depending on which scenarios a particular application uses.

Because the developed software systems are web applications, some of the identified components are for the client side and others for the server side. The client side was implemented using JavaScript in all three case studies. This results in similar and comparable code. However, the implementation of the components on the server side is language dependent, and they are only comparable at design level.

The code snippets are equal in all three cases. They are for the client layer. They are implemented in JavaScript and cover all the documented scenarios. They are encapsulated in a single file, which we consider to be the first step for building a usability components library.

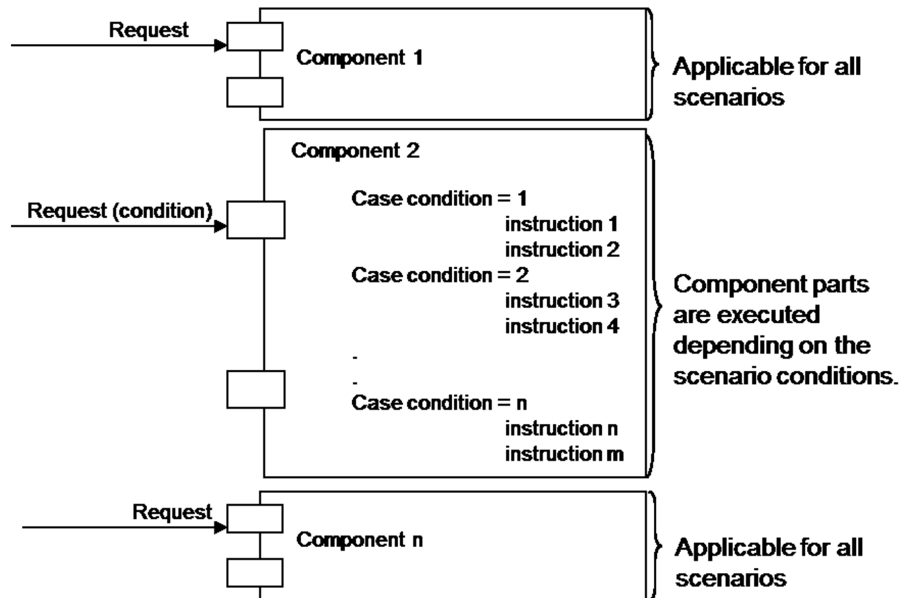

Fig. 9. Components and scenarios

8. Conclusions

In this paper, we explored the possibility of outputting reusable solutions for implementing three usability functionalities. From real implementations we found that there are three commonalities that can be generalized as reusable artefacts for different phases of the development process. The results of this study are confined to highly interactive web applications developed using the object-oriented paradigm. The results may differ for other types of applications.

The functionality covered by the reusable solution is confined to the application scenarios identified for the Abort Operation and Progress Feedback UMs and the requirements defined for the Preferences UM. New application scenarios or new requirements may emerge as new case studies are developed. It is useful to document scenarios and requirements using sequence diagrams from the very start of the development process. In the requirements elicitation and specification phase, these artefacts can be used to check that all the possible cases in which the usability functionality is applicable are taken into account. In the design phase, they are able to evaluate how the software system functionalities will be affected by the usability functionality and provide a clear of idea of how they should be implemented.

The proposed design pattern encapsulates all the functionality necessary to cover the responsibilities associated with each UM. The design will have to be modified according to the technology in which it is implemented, although we found that the client-side code is potentially common to any web application, as it uses a common script language (Javascript). Programming patterns are useful when the new implementation uses the

same programming language and the same program versions. The results specified as a design pattern are useful for implementing the solution in any programming language, whereas programming patterns provide useful code for other implementations or at least a guide for implementation in other programming languages.

The application of the patterns to other case studies developed by separate engineers identified faults in the documentation and the need to provide additional demo applications on top of the description of the code. Many of the reusable artefacts provided were found to be useful and, although it took longer to understand and learn the patterns first time round, they are potentially reusable in other implementations. One feature of patterns is that they are open to continuous improvement, and each new implementation will lead to upgrades, include other functionalities, improve the design and devise new useful code for other languages or versions.

Acknowledgments

This work 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

[1] L. Constantine and L. Lockwood, Software for use: A practical Guide to the Models and Methods of Usage-centered Design (Addison Wesley, New York, 1999).
[2] ISO, 9241-11, Ergonomic Requirements for Office Work with Visual Display Terminals. Part 11: Guidance on Usability, ISO, 1998.
[3] L. Trenner and J. Bawa, The Politics of Usability: A Practical Guide to Designing Usable Systems in Industry (Springer-Verlag, New York, 1998, 49-60).
[4] J. Battey, IBM's redesign results in a kinder, simpler Web site, 1999. http://interface.free.fr/Archives/IBM_redesign_results.pdf.
[5] G. Donahue, Usability and the Bottom Line, *IEEE Software*, 18(1) (2001) 31-37.
[6] J. Griffith, Online transactions rise after bank redesigns for usability, *Business Journal* (2002) http://www.bizjournals.com/twincities/stories/2002/12/09/focus3.html.
[7] J. Black, Usability Is Next to Profitability, *Bloomberg Business Week* (2003) http://www.businessweek.com/technology/content/dec2002/tc2002124_2181.htm.
[8] L. Bass and B. John, Supporting usability through software architecture, *Computer34(10) (IEEE, 2002)*113-115.
[9] D. Perry, A. Wolf, Foundations for the study of software architecture, *ACM Software Engineering Notes*, 17 (4) (1992) 40-52.
[10] L. Bass and B. John, Linking usability to software architecture patterns through general scenarios, *Journal of Systems and Software*, 66(3) (2003) 187-197.
[11] N. Juristo, A. Moreno and M.-I. Sanchez-Segura, Analysing the impact of usability on software design, *Journal of System and Software*, **80** (2007) 1506-1516.
[12] N. Juristo, A. Moreno and M.-I. Sanchez-Segura, Guidelines for Eliciting Usability Functionalities Software Engineering, *IEEE Transactions on Software Engineering*, **33** (2007) 744-758.
[13] N. Juristo, A. Moreno and M.-I. Sanchez-Segura, Usability Elicitation Patterns (USEPs) (2006) http://www.grise.upm.es/sites/extras/2/. Last visited: March 2012.

[14] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, Pattern - Oriented Software Architecture. A system of patterns (John Wiley & Sons, New York, 1996).

[15] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Patrones de Diseño. Elementos de software orientado a objetos reutilizable. (Addison-Wesley, 2003).

[16] D. Alur, J. Crupi, D. Malks, Core J2EE Patterns (Sun Microsystems Press Publisher, 2003)

[17] Pointer, PoInter: Patterns of INTERaction collection. http://www.comp.lancs.ac.uk/computing/research/cseg/projects/pointer/patterns.html. Last accessed: 2012.

[18] J. Tidwell, Designing Interfaces (O'Reilly Media, Inc., 2010).

[19] M. Welie and H. Trætteberg, Interaction patterns in user interfaces, in *Proc. of the Seventh Conference on Pattern Languages of Programming* (PloP), 2000.

[20] S. Laakso, User Interface Design Patterns, 2003. http://www.cs.helsinki.fi/u/salaakso/patterns/. Last accessed: 2012.

[21] The Usability Group at the University of Brighton, The Brighton Usability Pattern Collection, http://www.cmis.brighton.ac.uk/research/patterns/home.html. Last accessed: 2012.

[22] K. Perzel and D. Kane, Usability Patterns for Applications on the World Wide Web, in *Proc. of the Pattern Languages of Programming Conference*, 1999.

[23] D. van Duyne, J. Landay and J. Hong, The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience, (Addison-Wesley, 2002).

[24] Yahoo! Inc, Yahoo! Design Pattern Library, 2012. http://developer.yahoo.com/ypatterns/. Last accessed: 2012.

[25] A. Toxboe, User Interface Design Patterns Library, 2012. http://ui-patterns.com/patterns. Last accessed: 2012.

[26] Pattern Factory Oy, Patternry, 2012. http://patternry.com/. Last accessed: 2012.

[27] Infragistics, Quince. Interactive user experience (UX) design patterns library. http://quince.infragistics.com/. Last visited: 2012.

[28] M. Welie, Patterns in Interaction Design: The Amsterdam Collection, 2008. http://www.welie.com. Last visited: 2012.

[29] L. Bass, B. John and J. Kates, Achieving Usability through Software Architecture, in: Technical Report CMU/SEI-2001-TR-005, (ed. Software Eng. Inst., Carnegie Mellon Univ., 2001).

[30] B. John, L. Bass, M.-I. Sanchez-Segura and R. Adams, Bringing Usability Concerns to the Design of Software Architecture, in: *Proc. of EHCI-DSVIS'04: The 9th IFIP Working Conference on Engineering for Human-Computer Interaction and the 11th International Workshop on Design, Specification and Verification of Interactive Systems*, 2004.

[31] STATUS Project, Software Architecture that supports Usability, 2001. http://www.grise.upm.es/rearviewmirror/projects/status/index.html. Last visited: 2012.

[32] B. John, L. Bass, E. Golden and P. Stoll, A Responsibility-Based Pattern Language for Usability-Supporting Architectural Patterns, in Proc. of EICS, 2009.

[33] E. Folmer, M. Welie and J. Bosh, Bridging patterns: An approach to bridge gaps between SE and HCI, *Information and Software Technology*, **48** (2006) 69-89.

[34] M. Pinto, L. Fuentes, Aspect-Oriented Modeling of Quality Attributes, in Proc. of the Second European Conference on Software Architecture (ECSA), 2008, pp. 334-337.

[35] P. Runeson, M. Höst, Guidelines for Conducting and Reporting Case Study Research in Software Engineering, *Empirical Software Engineering*, **14** (2009) 131-164.