Software Design Guidelines for Usability

Laura Carvajal, Ana Moreno

Facultad de Informática - Universidad Politécnica de Madrid.
Campus de Montegancedo S/N, 28660. Madrid, Spain
lauraelena.carvajal@upm.es, ammoreno@fi.upm.es

Abstract. For years, the Human Computer Interaction (HCI) community has defined the expected characteristics of usable software systems. However, from a Software Engineering (SE) perspective, the process of incorporating usability into software is not always straightforward, as many usability features have strong implications in the underlying architecture. For example, successfully including a "cancel" feature in an application may involve complex interrelated data structures and functionalities. Our work is focused upon providing developers with a set of guidelines to assist them in including such usability features with high impact on software design into their developments effectively.

Keywords: Usability, software architecture, software design patterns

1 Introduction

For the better part of the last two decades, the HCI community has focused great effort in defining what makes a good user interface. Many solutions to common interaction problems have been proposed [1][2][3] yet, the SE community continues to struggle to consistently transform these solutions into actual software code [4].

There is a well documented gap between the contributions of both communities, namely the notions of what is expected of a usable application interface (HCI), and how applications should be crafted to achieve such expected levels of usability (SE) [4][5][6]. Works like [7][8][9][10][11] and [12] have attempted to bridge this gap, proposing solutions to include usability into systems from a software architecture point of view, yet the lack of traceability between them and each project's specific requirements is mostly overlooked. Furthermore, the proposed architectural solutions proposed thus far have consistently been of a high abstraction level [13], leaving another gap to be bridged between a general, high-level architectural solution and the manner in which developers are expected to translate it into actual software designs.

Our work focuses on addressing the present shortcomings and proposing detailed design solutions for a subset of HCI usability recommendations with proven impact on software architecture [14]. While there are many more usability scenarios that could be considered, we've chosen to focus our efforts on a set of usability features whose effects go beyond the GUI: those most relevant to software developers while always preserving the connection to the longstanding efforts and results of the HCI community, from which the chosen features all have been derived [15].

2 Related Works

There's been an extensive amount of research carried out in the past decade in regards to understanding and quantifying the strong relationship that exists between software architecture and usability. These results highlight the importance of that kind of a relationship and the need to address usability concerns from a software architecture perspective. Multiple approaches have been explored for addressing these concerns, mainly proposing diverse forms of architectural frameworks, guidelines and patterns in order to include usability into software systems correctly and effectively

The following studies represent the leading trend in the research area regarding usability support in software design. They have delved into determining the nature of the relationships that may exist between the usability needs of a software application and its architecture and propose solutions to addressing usability concerns at design time during application development.

Bass and John in [7] identify a set of usability scenarios that appear to have architectural implications, determine their potential usability benefits and propose software architectural (SA) patterns to help users realize those benefits.

In [8], Folmer et al. present an assessment technique, SLUTA, to assist software architects in designing the architecture of their systems in a way that supports usability. This technique promotes explicit evaluation of usability during architecture design, with the purpose of discovering usability issues during this early stage of development, as opposed to doing so during system maintenance to a higher cost.

In [9], Ferré et al. identify twenty usability patterns that, when present in a system, improve its usability. For each of these patterns and through the inductive process summarized below, the authors produce a possible design solution for incorporating them into the architecture of software applications.

Following the same research line as their 2002 work, in this later study by Bass and John [10] the authors introduce Usability Supporting Architectural Patterns (USAPs). Each USAP describes a usability concern, provides a set of responsibilities to be fulfilled, and describes an MVC-based sample solution for it, this time considering the forces, as defined by Alexander et al. [16] exerted over each scenario.

Seffah et al. in [11] identify and model specific scenarios that illustrate how internal software components may affect a system's usability. For each of the proposed scenarios, an existing or improved software design pattern is suggested as a potential solution to the scenario. These scenario-pattern pairs are ultimately documented and their application within a MVC architectural model is detailed.

In this most recent work by Bass and John [12] they alter the structure of the USAP and propose a pattern language based on software responsibilities, alongside a web-based tool for evaluating an architecture with respect to those patterns.

While the results obtained thus far are encouraging, there is still work to be done in this field. For instance, in most of the aforementioned works no empirical validation was performed, save for the three case studies carried out by Folmer, et al. [8].

Furthermore, the usability issues addressed in existing works as starting points are identified mostly by heuristic-based approaches. Ideally, the usability concerns to consider when proposing architectural and/or design patterns should be relevant from an HCI perspective and have proven implications on software architecture and design.

Most of the previous works deal with solutions at a high-level architectural level, which are not adequately validated. While architectural patterns can be very useful in depicting how a system should behave as a whole, our work explores the option of lower-level design patterns being more effective in detailing the responsibilities of its components.

In addition, none of the works studied provide any means of traceability between their proposed solutions and software requirements, which is of utmost importance for validation and maintenance purposes.

Therefore, we are presented with an open research problem related to providing users with efficient design and implementation artifacts to incorporate usability into a software system, and we intend to address it within the scope of this paper.

3 The Usability Guideline

The aim of this research is to provide software developers with recommendations to help them incorporate certain usability features into software systems. We have named these recommendations Usability Guidelines, from which Usability Design Guidelines are the main contribution of this work. These guidelines address the set of 11 usability issues with high implications on software functionality first identified in [14], and their full structure is shown in **Fig 1**.

Our hypotheses for this work are the following:

1. *The usability guidelines facilitate the inclusion of functional usability features into software designs*. This will be measured and contrasted in terms of over-all and by-guideline design time (quantitative) and perceived ease of use of the guidelines (qualitative).
2. *The usability guidelines improve the quality of the designs, resulting in better software*. We will argue that the usage of the proposed guidelines produces designs with higher levels of a closed set of quality design attributes when compared to equivalent projects that do not make use of the proposed guidelines or that only use it partially.

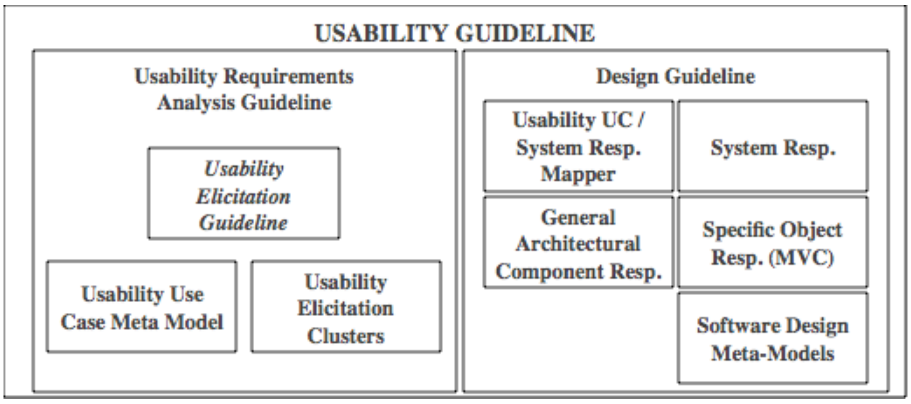
USABILITY GUIDELINE
Usability Requirements Analysis Guideline
Usability Elicitation Guideline
Usability Use Case Meta Model
Usability Elicitation Clusters
Design Guideline
Usability UC / System Resp. Mapper
System Resp.
General Architectural Component Resp.
Specific Object Resp. (MVC)
Software Design Meta-Models

Fig 1 Structure of the Usability Guideline. Comprised of Usability Requirements Analysis Guideline and Usability Design Guideline.

The Usability Requirements Analysis guideline is made up of the following three components:

1. **The Usability Elicitation Guideline (UEG)**, an existing contribution by [15] extended for this work, whose aim is to help in eliciting usability requirements
2. **The Usability Elicitation Clusters**, a graphic representation of the Usability Elicitation Guideline, designed partly to help analysts understand the flow of the requirements discussion items
3. **The Usability Use Cases Meta-model**, a use case representation of the usability needs covered by the UEG to help designers include them in their use case models.

The Usability Design Guideline is comprised of five parts, namely:

1. **The System Responsibilities**, or the main functionalities that the system should accomplish in order to fulfill all of what has been elicited with the UEG.
2. **The Usability Use Cases / System Responsibilities Mapper**, to help determine which System Responsibilities apply in relation to the Usability Use Cases.
3. **The Generic Architectural Component Responsibilities**, describe the System Responsibilities at the lower abstraction level of generic architectural components.
4. **The Specific Object Responsibilities (for MVC)**, which instantiates those Architectural Component Responsibilities for a specific architecture (MVC).
5. **The Software Design Meta-models, which are the** graphic representation, as class and sequence diagrams, of the Specific Object Responsibilities.

The core of the contribution of this work is represented by the Usability Design Guideline as an answer to the need for supporting development teams in including functional usability features during the design phase of their projects. However, the Usability Requirements Analysis Guideline also embodies an important part of our contribution in regards of complimenting the original UEGs and establishing a needed 'bridge' between the artifacts provided for both the analysis and design phases.

We have developed such guideless for the eleven usability mechanisms detailed in [13]. The next section shows an example for one of the mechanisms.

4 Sample Usability Guideline: Abort

The Abort Functional Usability Feature covers providing the means to cancel tasks and allowing exiting the application altogether. When tasks take a long time to execute the user might want to abort them. They must also be allowed to exit the application at any time, properly handling any on-going tasks.

Below is an overview of the two parts of the Usability Guideline for this feature: the Usability Requirements Analysis Guideline and the Usability Design Guideline.

4.1 Usability Requirements Analysis Guideline

The Usability Requirements Guideline is made up of four artifacts: the Usability Elicitation Guideline, the Usability Elicitation Clusters, the Usability Use Case Meta-models and the Usability Use Case Dependencies Mapper. Reduced versions of each of these artifacts are presented below for the Abort feature.

4.1.1 Usability Elicitation Guideline

The Usability Elicitation Guideline explains the basics of Abort functionality, including which of the system's actions need to be cancellable, what state the system will return to after cancellation and how to handle on-going actions upon

Table 1 shows a partial view of this Usability Elicitation Guideline: one of the two HCI recommendations that comprise it, namely the one titled "Cancelling Commands", which deals with aborting ongoing commands, and the effects this may have on the application state and how to handle them.

Table 1. Usability Elicitation Guideline. Part 1 of 4 of the Usability Requirements Analysis Guideline for the Abort feature.

Identification			
Name	**Abort**		
Family	Undo/Cancel		
Intent			
Providing the means to cancel an on-going task, or to allow for exiting the application altogether			
Problem			
Certain tasks might take a long time to execute. In such cases, the user will need to be at the liberty to cancel them. S/he must also be allowed to exit an application at all times, regardless of any tasks that may be being executed.			
Context			
When the user needs to exit an application or a command quickly.			
Interrelationships			
When implementing the Abort feature, **Undo** functionality will be needed for the cancellation of commands, in order for the application state to be properly reverted. Also, if implementing an application that prompts the user to save changes upon exiting, parts of the **Warning** feature will be needed.			
HCI Recommendation	**Elaboration**	**Discussions with Stakeholders**	**Usage Examples (opt)**
A_HCI-1 Cancelling Commands If a command takes over 10 seconds to execute, a 'cancel' option must be provided, interrupting execution and correctly handling the resulting system state.	**SBS_ELAB-1 Back and Cancel** A 'command' is understood as an indivisible unit of execution. All longer commands (>10s) must be identified. The state that the system must revert to, if any, must be determined for each.	**U_HCI-1:** Which commands will require a cancel option? **U_HCI-2:** For all cancelable commands, how should the cancel option be presented to the user? **U_HCI-3:** For all cancellable commands, which state will the system go to after the user chooses the cancel option?	**A_EX-1 Exporting Video File In Apple's Quicktime,** when choosing the option to 'export' a video file into a different format, the application does so presenting a progress bar with a cancel button. Upon cancellation, any portion of the video that was exported is automatically sent to the 'Trash'.

4.1.2 Usability Elicitation Clusters

From the partial view of the Usability Elicitation Guidieline in Fig 1, two Elicitation Clusters can be generated as shown in Fig 2; A_EC-1 and A_EC-2, which group all the discussions related to cancelling commands and handling system state.

These elicitation clusters will give way to the System Responsibilities of the Usability Design Guideline for the Abort Functional Usability Feature as seen below.

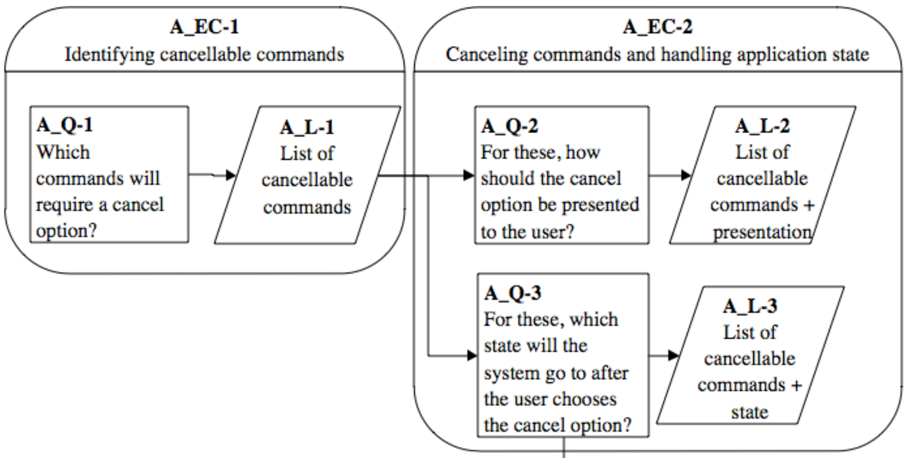
A_EC-1
Identifying cancellable commands
A_Q-1
Which commands will require a cancel option?
A_L-1
List of cancellable commands
A_EC-2
Canceling commands and handling application state
A_Q-2
For these, how should the cancel option be presented to the user?
A_L-2
List of cancellable commands + presentation
A_Q-3
For these, which state will the system go to after the user chooses the cancel option?
A_L-3
List of cancellable commands + state

Fig 2 Usability Elicitation Clusters (partial view) Part 2 of 4 of the Usability Requirements Analysis Guideline

4.1.3 Usability Use Case Meta-model

The Use Case Meta-model for the Abort Functional Usability Feature is shown in **Fig 3** in which eight use cases are identified. Four of these use cases are *borrowed* use cases: W_UC_3 from the Warning feature, SPF_UC-1 from the Progress feature, U_UC-1 and U_UC-6 from the Undo feature (dark gray, all outside of the scope of this paper). The remaining four use cases are comprised of three *concrete* use cases (light gray, to be used directly within the final use case model of a project) and one *template* use case (white, to be instantiated by the appropriate, project-specific use case), all solely belonging to the Abort mechanism itself.

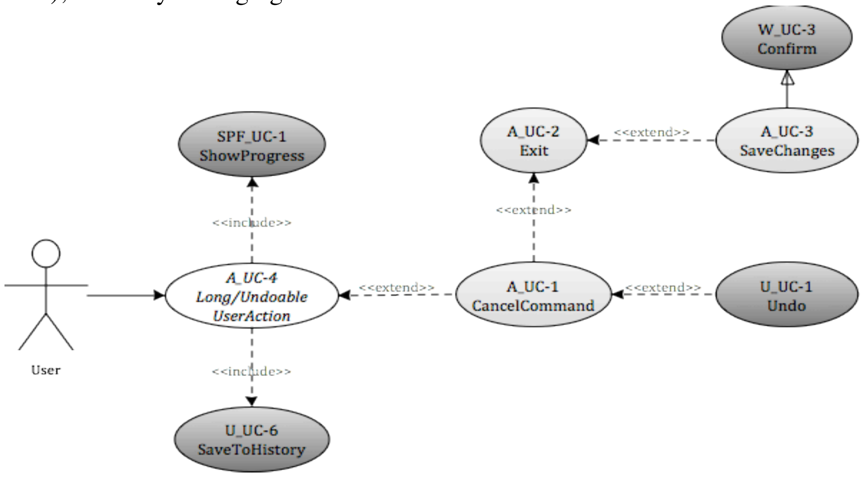

Fig 3. Usability Use Case Meta-model. Part 3 of 4 of the Usability Requirements Analysis Guideline.

The applicability of each of these use cases will depend on the results of the elicitation process. If during elicitation of the Abort Functional Usability Feature it is determined that, for example, exiting the application will never imply saving changes, the A_UC-3 and U_UC-3 use cases will be discarded at modeling time.

4.2 Usability Design Guideline

The Usability Design Guideline is comprised of five main parts, described below: The general System Responsibilities that must be fulfilled in order to fully implement the Abort Functional Usability Feature, the relationships between the System Responsibilities and the Usability Use Cases, the Generic Architectural Component Responsibilities, the Concrete Object Responsibilities (for a MVC architecture) and the Usability Design Meta-models for those Concrete Object Responsibilities as object-oriented class and sequence diagrams.

4.2.1 System Responsibilities

From the Elicitation Clusters identified above, four System Responsibilities are derived, for which a partial view is offered in Table 2. More specifically, the System Responsibilities A_SR-1 and A_SR-2 shown below are yielded directly by the Elicitation Clusters shown in **Fig 2**, A_EC-1 and A_EC-2.

Table 2. System Responsibilities List. Part 1 of 5 of the Usability Design Guideline.

System Responsibilities List for Abort	
A_SR-1	**Identify cancellable commands:** The system must keep track of commands that are cancellable
A_SR-2	**Cancel commands and handle application state:** The system must allow users to cancel (cancellable) commands and to handle app state appropriately

4.2.2 Use Case / System Responsibilities Mapping

The Usability Use Case / System Responsibilities Mapper depicts which System Responsibilities are related to which Usability Use Cases. Responsibilities in the right column will *not* be present in the resulting system if the use case in the left column is not contemplated at elicitation time.

Table 3 shows how the System Responsibilities for the Abort feature depend on the feature's use cases.

Table 3. Use Case/System Responsibilities Mapping. Part 2 of 5 of the Usability Design Guideline for the Abort feature.

Use Cases	**Dependent Responsibilities**
A_UC-1 Cancel	A_SR-1 Identify cancellable commands A_SR-2 Cancel commands and handle application state
A_UC-2 Exit	A_SR-3 Exit application handling potential on-going commands A_SR-4 Handle potential changes to be saved
A_UC-3 Save Changes	A_SR-4 Handle potential changes to be saved
A_UC-4 Long/Undoable User Action	A_SR-1 Identify cancellable commands A_SR-2 Cancel commands and handle application state

4.2.3 Generic Component Responsibilities

Table 4 shows a partial view (two of four System Responsibilities) of the suggested Generic Architectural Components for the Abort Functional Usability Feature.

Table 4. Partial view of the Generic Component Responsibilities. Part 3 of 5 of the Usability Design Guideline for the Abort feature.

System Responsibility	**Generic Component Responsibilities**	
A_SR-1 Identify and execute cancellable commands	**A_CR-1**	A software component, preferably that responsible for handling user events (*UI*), must know of all the commands that are cancellable. By being in charge of this responsibility, it will be able to display the necessary interface components to provide the user with the means to cancel the command.
	A_CR-2	The *UI* is also responsible for listening for command invocations from the user.
	A_CR-3	Execution of actions is always the responsibility of the pertinent Domain Component in the application
	A_CR-4	The component in charge of delegating actions (if any) should determine whether the action is undoable or not, from a pre-established list.
	A_CR-5	If the action to execute is undoable, it must first be encapsulated as an instance of a Command Component, together with any pertinent state information and the necessary actions needed to revert its effects.
	A_CR-6	Such an instance is then stored in a History Component, responsible for keeping a single (ordered) collection of all executed undoable actions.
	A_CR-7	After encapsulation, the Domain Component is free to execute the invoked action

4.2.4 Concrete Object Responsibilities for MVC

When instantiating for an MVC architecture, the generic architectural components described above can be translated into the following system objects (see **Table 5**).

Table 5. Partial view of the Concrete Object Responsibilities for MVC for the Abort feature. Part 4 of 5 of the Usability Design Guideline.

System Resp.	**Objects**				
	View	**Controller**	**ConcreteComm.**	**HistoryList**	***DomainClass***
A_SR-1 Identify and execute cancellable commands	1. The *View* must listen for calls to commands. It must be aware of which of these are cancellable and provide the appropriate GUI components to enable cancellation. (A_CR-1)	▨	▨	▨	▨
	1. The *View* must listen for invocation of actions. Upon reception, it must notify the *Controller* of the action (A_CR-2)	2. The *Controller* must determine if the invoked action is cancellable. In such case it execute the corresponding *Concrete Command* object, otherwise the call goes directly to the *DomainClas* (A_CR-4) 3. The *Controller* must then add the *Concrete Command* to the *HistoryList*.	4a. Upon execution, the *Concrete Command* saves the state information and calls the appropriate action in the corresponding *DomainClass* (A_CR-5)	4b. The *HistoryList* saves the cloned *ConcreteCo mmand* atop its collection (so it can later be available to undo) (A_CR-6)	5a. The *DomainClass* executes the appropriate method to carry out what was originally invoked by the user through the *View*. (A_CR-3, A_CR-7)

4.2.5 Usability Software Design Meta-models

These UML diagrams represent the Concrete Object Responsibilities described in the previous sections. The following subsections describe the class diagram and the classes involved in the Abort feature and their interrelationships, followed by the description of one the three sequence diagrams that fully flesh out the present feature.

Class Diagram
Fig 4 below shows the class diagram for the Abort Functional Usability Feature. As described in the Concrete Object Responsibilities Table, the main objects involved are the View, Controller, HistoryList, ConcreteCommand and DomainClass. The first two, fulfilling their role within MVC, respectively capture and distribute the user calls to perform actions. The HistoryList stores invoked commands, ConcreteCommand implements a Command interface, as described by [17], and is responsible for ordering the execution of a requested action (in DomainClass) as well as for storing all state information required for eventually undoing the command it represents (i.e the method it's calling in DomainClass)

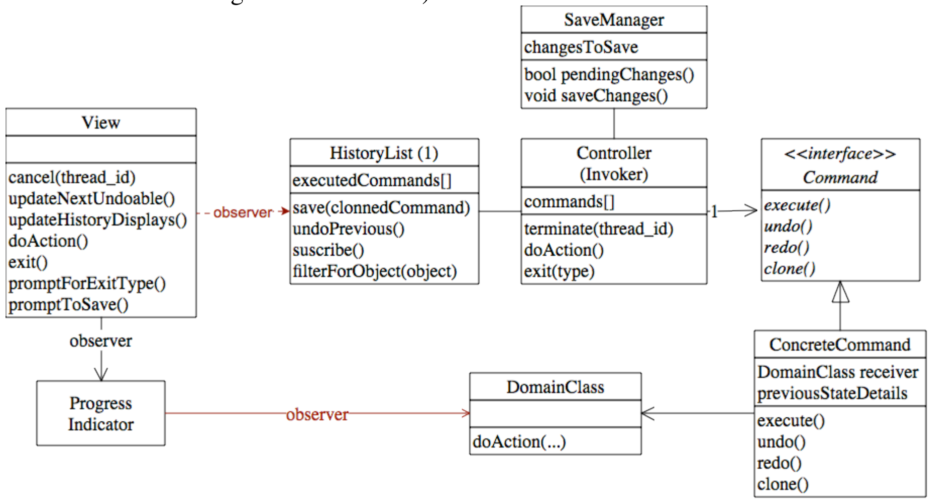

Fig 4 Usability Software Design Meta Model. Class Diagram. Part 5a of 5 of the Usability Requirements Analysis Guideline.

Sequence Diagram "Cancel Command"
The sequence diagram for cancelling a command is shown in **Fig 5**. This is the UML representation of the object responsibilities and sequences presented in **Table 5**. It starts when the user requests to cancel an on-going command. The View has the information that identifies the command and passes it onto the Controller. With it, the Controller finds the thread that the command is running in and orders it to stop. It then orders the HistoryList to undo whatever changes were produced by that command while it ran. The HistoryList orders the corresponding ConcreteCommand to undo, which leads to the DomainClass reverting the state to what it was before the execution of the command. Once the effects have been reverted, the Controller orders the thread to end, sending a notification to any subscribed Progress indicators, which proceed to terminate. Finally, the GUI updates to reflect the command has been cancelled.

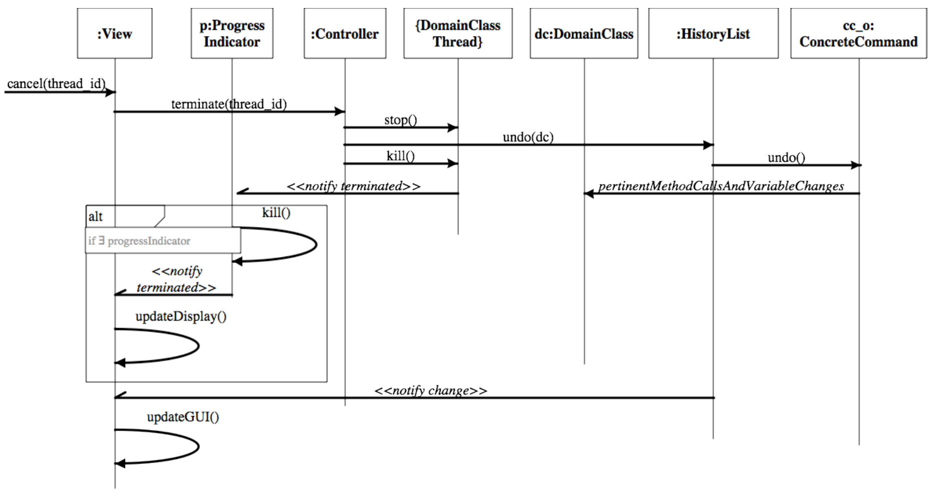

Fig 5 Usability Software Design Meta Model. 'Cancel Command' Sequence Diagram. Part 5b of 5 of the Usability Requirements Analysis Guideline.

The guidelines developed for the different mechanisms where applied in different projects as we will discuss in next section.

5 Results

The proposed guidelines were applied during construction of three different software projects by nine teams (three teams per project) of Software Engineering master's students at the UPM School of Computing. These projects were:

1. **An online list management system**: An application to manage to-do lists with the possibility of sharing and scheduling tasks, as well as organizing them visually.
2. **A console for a home automation system**: An application to operate a simulated network of sensors and actuators that controlled various features of a home environment (lights, air conditioner, blinds, garage door, etc) in real time.
3. **An auction site**: A web application with basic auction functionalities.

Each of the three teams (numbered PiG1, PiG2, PiG3, where 'i' represents the project number from the above list) was provided with the same Software Requirements Specifications (SRS). The differencing factor was the type of help that was provided for each team:

- The first team in each group is given the full Usability Guidelines for each of the features that apply to their project.
- The second team in each group is given a partial guideline for every needed feature. This partial guideline contains the full Usability Requirements Analysis Guideline and only one element of the Usability Design Guideline, namely the System Responsibilities. In other words, the Generic Component Responsibilities, the Concrete Object Responsibilities and the Usability Software Design Meta-models were not given to these teams for any of the features.

– The third team in each group was not given any part of the guideline

The three SRSs already included usability requirements that were marked as pertaining to one of the 11 Functional Usability Features in this work. Each team was expected to design and implement their full SRS including all usability aspects.

Upon submission, the Software Usability Guidelines were being evaluated both in a qualitative and a quantitative manner. The quantitative evaluation is still under way at the time this writing, but partial results of the qualitative evaluation are shown in Fig 6[1] below. A broader array of earlier results is presented in [18], excluded herein due to space restrictions.

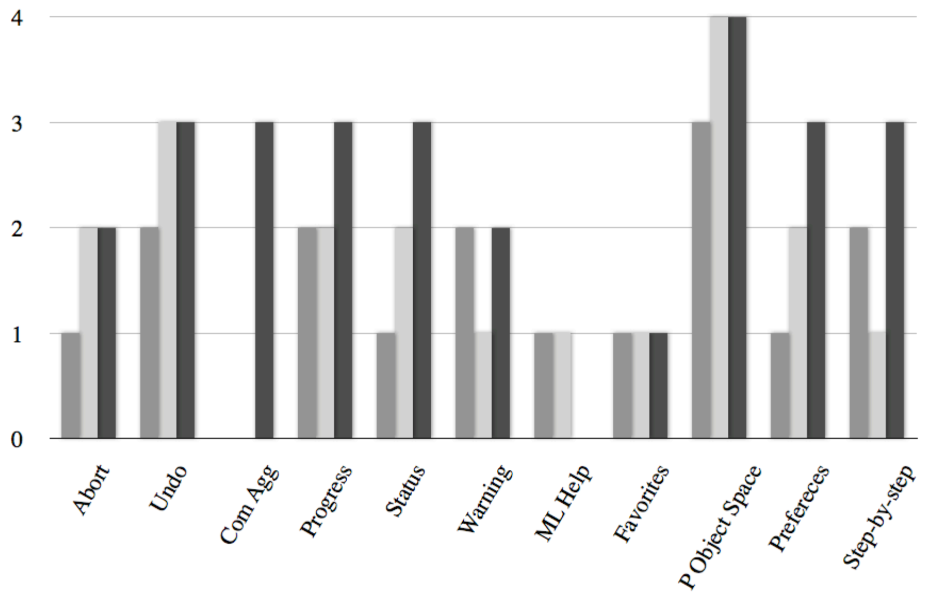

Fig 6 Responses to question "How would you rank this feature in terms of the complexity you encountered during the: a) design phase?" [Legend: medium gray = with full guideline. Light gray = with partial guideline. Black = with no guideline].

As an example, **Fig 7** and Fig 8 show the way in which the Abort mechanism was applied by one of the development teams. These figures depict the project's class diagram (partial) and one sequence diagram, respectively. They focus on the system requirement pertaining cancellation of the opening or closing of window blinds, which involves stopping their movement and returning the blinds to their original state.

[1] In Figure 6, the missing data points indicate that the corresponding feature was not a part of the project being graphed.

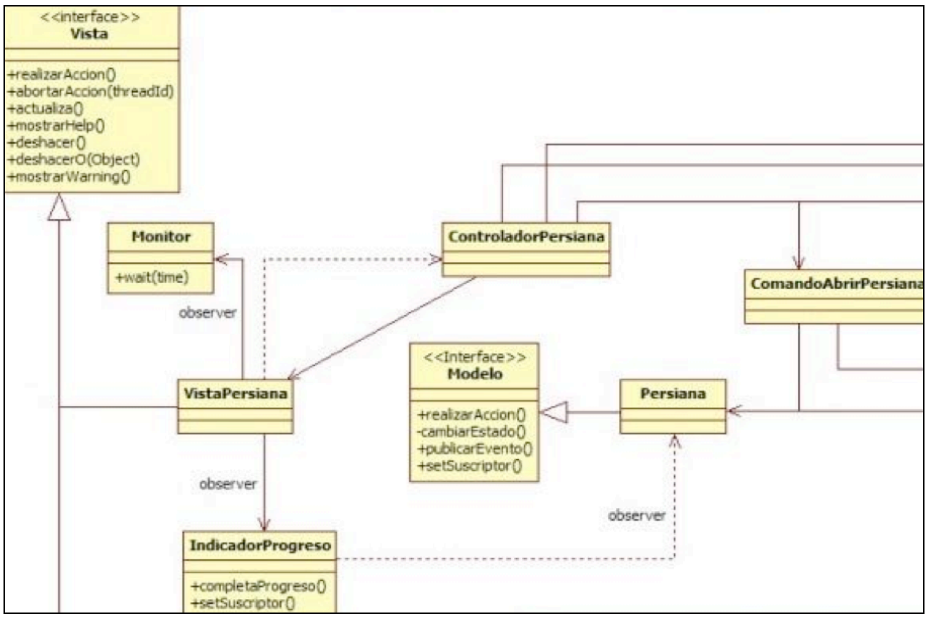

Fig 7 Partial view of the team's class diagram

In **Fig 7** above, we can see the teams representations of the classes proposed by the Abort Guideline in **Fig 4**: The View (Vista), ProgressIndicator (IndicadorProgreso), DomainClass (Persiana), Controller (ControladorPersiana) and ConcreteCommand (ComandoAbrirPersiana). HistoryList (HistorialUndo) is also present in the diagram but cut out of frame for space reasons.

In Fig 8 below, we can see the team's collaboration diagram[2], where the proposed interactions of the Abort Guideline for cancelling a command (**Fig 5**) are presented. In this case, the command to cancel is the opening/closing of window blinds, where the View orders the Controller (ControladorPersiana) to stop the thread responsible, then orders the HistoryList (HistorialUndo) to undo any effects (causing the DomainClass Persiana to ultimately revert any movement of the blinds made prior to cancellation). Once control is returned to ControladorPersiana it kills the thread and the view has since been notified of the change and updates the GUI accordingly.

[2] P2G1 represented their interaction diagrams as collaboration diagrams for presentation reasons.

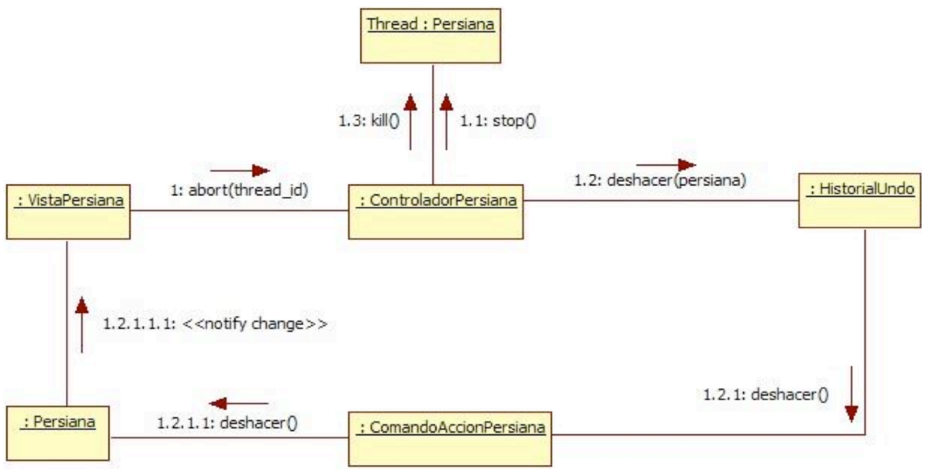

Fig 8 Team P2G1's collaboration diagram, showing the interactions suggested by the sequence diagram in Fig 5, instantiated for the window blinds use case in the Home Automation Project

6 Conclusions and future work

In this paper we present one possible solution to guide software developers in including eleven usability features involving software architecture into their applications.

Preliminary validation results are encouraging. On the qualitative front, they show that the guidelines help reduce the perceived complexity of the functional usability features. Also, early quantitative validation results show that development time is reduced when development teams make use of the guidelines proposed herein

While there are other usability features not considered in this work that could potentially impact software functionality, our solution provides an important contribution in the SE and HCI fields. The emphasis on assisting developers address usability features with the highest impact in system functionality remains paramount.

The proposed guidelines help software development teams to include usability features from the earliest phases of development all the way through software design. Further research should focus on providing artifacts that would aid the implementation phase. Providing developers with software plug-ins, for example, would further aid a smooth inclusion of usability features into software.

Furthermore, at the time of this writing a tool is being developed to automate the use of the guidelines by developers, which should facilitate its use and ultimately further improve results.

7 References

[1] Nielsen, J. 1993. Usability engineering. Boston, Massachusetts, USA: Morgan Kaufmann.

[2] L. Constantine, L. Lockwood. Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design. Addison-Wesley, 1999.

[3] M. van Welie. The Amsterdam Collection of Patterns in User Interface Design. Welie.com

[4] Seffah, A. M. 2004. The obstacles and myths of usability and software engineering. Commun ACM, 47 12, 71-76.

[5] Constantine, L., Biddle, R., Noble, J. Usage-Centered Design and Software Engineering: Models for Integration. Bridging the Gaps between Software Engineering and Human-Computer Interaction. 106-113. 2003.

[6] Walenstein, A. Finding Boundary Objects in SE and HCI: An Approach Through Engineering-oriented Design Theories. Bridging the Gaps between Software Engineering and Human-Computer Interaction. 92-99. 2003.

[7] Bass, L., & John, B. E. 2002. Linking Usability to Software Architecture Patterns Through General Scenarios. J Syst Software, 188-197.

[8] Folmer, E., van Gurp, J., & Bosch, J. 2005. Software Architecture Analysis of Usability. Lect Notes in Comput Sc, 3425, 38-58.

[9] Ferre, X., Jusisto, N., Moreno, A., & Sánchez, M. 2003. A software architectural view of usability patterns. Proceedings of INTERACT 2003.

[10] John, B., Bass, L. & Sanchez-Segura, M. 2005. Bringing Usability Concerns to the Design of Software Architecture. Lecture Notes in Computer Science, 3425, 1-19.

[11] Seffah, A., Mohamed, T., Habieb-Mammar, H., & Abran, A. 2008. Reconciling usability and interactive system architecture using patterns. J Syst Software

[12] Bass, L., John, B. E., Golden, E., Stoll, P. A Responsibility-Based Pattern Language for Usability-Supporting Architectural Patterns. EICS'09, July 15–17, 2009, Pittsburgh, USA.

[13] Carvajal, L., 2009 Usability-enabling Design Guidelines: A design pattern and plug-in solution. Proceedings of the doctoral symposium for ESEC/FSE. No. 42

[14] Juristo, N., Moreno, A., & Sanchez-Segura, M. 2007. Analysing the impact of usability on software design. J Syst Software, 1507-1516.

[15] Juristo, N., Moreno, A., & Sanchez-Segura, M.I. 2007. Guidelines for eliciting usability functionalities. IEEE Transactions on Software Engineering, 744-758.

[16] Alexander, C., Ishikawa, S., and Silvernstein, M. A Pattern Language, Oxford University Press, New York, 1997.

[17] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns Elements of Reusable Object-Orientated Software., Addison -Wesley.

[18] Carvajal, L., Moreno, A., Usability-enabling design patterns. IADIS International Conference Interfaces and Human Computer Interaction 2010