Addressing Usability Requirements in Mobile Software Development
Addressing Usability Requirements in Mobile Software Development

Rafael Capilla, Laura Carvajal, Hui Lin

Abstract: Usability is an important quality requirement for many of the today software applications, where the complexity of modern user interfaces and the quick reaction required by users demanding highly usable software. However, usability is many times poorly addressed as usability requirements often require an additional implementation effort. In addition, usability is sometimes rarely described in the software architecture so it is difficult to perceive where and how usability impacts in the software design. In this chapter we analyze the impact of two usability mechanisms in the software architecture of a mobile software application, as this kind of software demand stringent usability requirements. We also map the generic architectural responsibilities of the usability mechanisms analyzed to concrete classes in the software architecture of the mobile application supporting such functionality.

Keywords: usability, human computer interaction (HCI), mobile software, software architecture.

1. Introduction

The software architecture community recognizes the importance of quality attributes for both software design and implementation methods, but such qualities are many times neglected and poorly documented. Software quality evaluation methods (such as ATAM or ARID [5]) are often used to evaluate the quality of the architecture in the early stages of the software design process.

Nowadays, one key relevant quality attribute for modern software development is usability as, since the eighties, usability features have played a key role for building more usable software. Usability has been defined in various ways in literature throughout the years. One of the most widely known definitions, proposed by the ISO 9126 standard, (replaced by the standard ISO/IEC 25000:2005 – http://www.iso.org) describes usability as “the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions”. Other authors [16] [21] recognize the importance of usability in multiple factors and the difficulty to put usability in practice, but the majority of the proposed definitions consider usability as a critical aspect for interactive systems. Such relevance has been highlighted in several studies [4] [6] [18] aimed to
demonstrate the benefits of usability’s, including reduction of documentation and training costs, improvement of productivity, increase in morale and e-commerce revenue, and more. Accordingly, large-scale companies like IBM and Boeing Co. have begun to consider usability as a key factor when developing and buying software.

Nonetheless, despite the obvious potential rewards, we keep falling short of producing usable software, and even when developers are convinced of their need to introduce usability features, this process is far from straightforward. One of the main reasons behind these difficulties is the lack of integration between the two communities involved in the process of building usable software, namely the Human Computer Interaction (HCI) and Software Engineering (SE) communities. Generally speaking, the former knows which characteristics have to be included in a software system to make it usable and the latter knows of how to build software correctly. The differences in terminology and development approach used by both communities lead to well-known HCI practices that are not integrated in SE common practices.

In addition, some application domains like the mobile software demand stringent software requirements and usability mechanisms tailored for this kind of apps. Challenges related to the size of the interface, performance, notification to users when the mobile loses the connection, and customized text entry among others, often drive the selection and use of usability mechanisms specific for mobile software. At present, there are a number of challenges related to the integration of usability features in common SE practices and where software designers demand assessment guidelines on how to introduce and describe usability mechanisms both in the architecture and code.

The remainder of this chapter is as follows. Section 2 discusses the related work. In section 3 we describe a taxonomy of usability mechanisms that can be applied for mobile software development. Sections 4 and 5 outline the use cases and major architectural responsibilities of two usability mechanisms presented. In section 6 we analyze the impact of the two usability mechanisms selected in the architecture of a mobile application, and we discuss the implications at the design level. Finally, section 7 provides a discussion of the presented approach and in section 8 we draw the conclusions of our work.

2. Related Work

In order to understand the relationship between usability and software design, we describe in this section experiences and related work between usability and software architecture and related experiences in the mobile applications domain.

One early work [1] describes 27 relevant usability scenarios and proposes a solution to design them in the form of architectural patterns. The authors later evolved this work in [11], introducing what they termed Usability Supporting Architectural Patterns, which propose MVC-specific sample solutions for usability
concerns in UML. Some years later, the authors in [12], looked to test this solution in industry. In their experiment, the authors realized that the general response was a resistance to the use of UML-based solutions. Such feedback led them to remove the UML diagrams and replace them with textual descriptions explaining the structure and behavior of the solution, without imposing a particular architecture. Complementarily, in [7] the authors identified twenty usability patterns and yielded a design solution for incorporating them into the architecture of software applications. A different approach [22] suggests existing or improved software design patterns as potential solutions to specific scenarios that illustrate how internal software components may affect a system’s usability.

Today, the importance of mobile software in many application domains require stringent software requirements that need more and more usability attributes in order to facilitate the usage by end-users (e.g.: quick reaction, interfaces easy to use, adequate feedback). Consequently, usability becomes a major concern for modern mobile applications for building highly usable interfaces. Several studies and experiences highlight the role of usability in mobile phone applications, and the evaluation and effects of usability might be different in a usability versus a real environment [2] [13], because if the mobile user depends on location properties, physical motion [15], and specific contexts (e.g.: mobile context) the effectiveness and efficiency as a quality of use in a particular environment [3] [14] [20] [23] may vary. Moreover, the evaluation and study of human computer interaction in mobile devices have been analyzed in [8] [10] in order to identify usability problems with apps installed on mobile phones (e.g.: enlarge or minimize the interface). Other works [19] have reported the results of the study of usability features that influence the feedback of user tasks in different scenarios as such results impact of the design of the target system. Furthermore, the authors in [17] analyze usability concerns in mobile apps for the NFC (Near Field Communication) technology, where usability is not only implemented in terms of usable interfaces but also in the form as specific usability mechanisms such as a Progress Bar, which increases user satisfaction as it provides the necessary feedback during user interaction with the mobile device.

The majority of the approaches describing the relationship between architecture and usability only provide guidelines and recommendations to introduce usability patterns or mechanisms but not the details on how to perform this task or how to introduce such mechanism in concrete architectural styles. The approach described in this chapter goes a step beyond as we define the generic and concrete architecture responsibilities for three-layered architectures and based on our previous experience using the MVC pattern. Moreover, most usability research aimed to analyze the impact of usability features in mobile apps has focused on testing and evaluating the effects of usability attributes, such as: context, flexible interfaces, display different resolutions, and data entry methods among others, but they don’t describe the impact of usability requirements in the software architecture and how to implement such mechanisms in the architecture of mobile systems.
3. Usability Mechanisms for Mobile Applications

Based on the lacks summarized in the related work we analyze the usability mechanisms that are of special interest for mobile applications in order to help software developers to introduce usability mechanisms into the software architecture. Our study focused on 5 PDA and Smart-phone mobile applications developed at the Rey Juan Carlos University between 2005 and 2011 for which we analyzed the usability features implemented on them. We also base our analysis on our previous experience for classifying usability mechanisms for the MVC pattern. As a result, we provide the following classification of usability mechanisms that we believe is suitable for mobile apps and we include examples of use.

**System Status Feedback usability mechanism:** Inform users about the internal status of the system
**Mechanisms/functionality implemented:** Status Display.
**Examples:** A status icon or bar when a web page is being accessed using a Smartphone. A status window to notify that a resource is not available (e.g.: a Bluetooth device). Check if a connection is available before doing something. Show an icon indicating connection.

**Interaction usability mechanism:** Inform users that the system has registered a user interaction, i.e. that the system has heard users.
**Mechanisms/functionality implemented:** Interaction Feedback, Let User Know What’s Going On.
**Examples:** The mobile shakes when accessing an application.

**Warning usability mechanism:** Inform users of any action with important consequences.
**Mechanisms/functionality implemented:** Warning.
**Examples:** Warn about an unsecure web site or inform about removing a data from the agenda.

**Long Action Feedback usability mechanism:** Inform users that the system is processing an action that will take some time to complete.
**Mechanisms/functionality implemented:** Progress Indicator. Let User Know What’s Going On.
**Examples:** A bar indicating progress, an icon with a timer, a message displaying the remaining time to complete a task (e.g., progress connecting to a service).

**Global Undo usability mechanism:** Undo system actions at several levels
**Mechanisms/functionality implemented:** Go Back One Step
**Examples:** A mobile web app including a questionnaire that allows you to reset values.
Abort Operation usability mechanism: Cancel the execution of an action or the whole application.
Examples: A mobile web app including a functionality to fill several questionnaires that allows you to cancel the process while filling any questionnaire going back to a specific place. Cancel sending a file.

Go Back usability mechanism: Go back to a particular state in a command execution sequence.
Mechanisms/functionality implemented: Go Back to a Safe Place, Go Back One Step.
Examples: Home button, Back button.

Structured Text Entry usability mechanism: Prevent users from data input errors.
Examples: Forms. Reduce the number of screens to introduce and display data. A Phone agenda.

Step-by-step execution usability mechanism: Help users doing tasks that require different steps with user input and correct such input.
Mechanisms/functionality implemented: Step-by-Step
Examples: Configuration of a new mobile application which is downloaded and installed in the device.

Preferences usability mechanism: Record each user's option.
Mechanisms/functionality implemented: User preferences.
Examples: Configure mobile phone settings or customized settings in a mobile app.

Favourite’s usability mechanism: Record certain places of interest for the user.
Mechanisms/functionality implemented: Favourites.
Examples: Favourites web pages only for mobile apps using a web interface.

Help usability mechanism: Provide different help levels for different users.
Mechanisms/functionality implemented: 1-help level.
Examples: In contrast to standalone applications where multi-level help is common, small interfaces like those used in mobile applications often implement 1-help level.

From our previous classification on usability mechanisms that can be employed in the development of mobile software we analyze the use and impact of such us-
ability mechanisms in the design process that we formulate in terms on the follow-
ing research questions (RQs):

RQ1: Which are the responsibilities of usability mechanisms from the software architecture point of view?

RQ2: How usability mechanisms impact on the architecture of mobile applications?

In this research we will focus only in two usability mechanisms for which we believe they represent important usability concerns demanded by modern mobile software apps. The two usability requirements we will study are:

a) **System Status Feedback**: Many mobile applications should inform users about the status of an action and let users know about the status of a given task, such as for instance using a message or a warning.

b) **User Preferences**: All modern mobile devices include support to configure user preferences about a variety of configurable options such as: wireless and radio settings, display, sound, privacy, energy consumption, etc. Therefore, users are allowed to check and change the current phone configuration and enable/disable some settings of the applications installed.

In the following sections we outline how the aforementioned usability mechanism impact on the software architecture. We describe the functionality of each usability mechanism through use cases and we map each use case to generic and concrete architectural elements that implement such functionality.

### 4. System Status Feedback

In this section we focus on the System Status Feedback (SSF) mechanism. According to Table 1 the status display of a mobile application can warn users about the availability of a resource. More specifically, the use cases we identified for the system status feedback (SSF) mechanism are described in Figure 1:

**SSF_UC-1 Change Status**: A status change can be triggered by a user and he/she must be aware of system status and its changes.

**SSF_UC-2 Handle User-initiated Status Changes**: Changes in the status initiated by the user must be handle by a specific component and notified the results to the parties involved. A status manager often takes care of such status changes and updates the status information.

**SSF_UC-3 Handle System-initiated Status Changes**: In some cases changes its status and this action must be handled similarly like in the SSF_UC-2 use case.
**SSF_UC-4 Display Status**: Changes in the status must be displayed and notified to users.

![System Status Feedback Use Case Model]

**Figure 1: System Status Feedback Use Case Model**

### 4.1 SSF generic component responsibilities

According to the use cases of Figure 1, the description of the major usability design guidelines and the generic component responsibilities (SSF_SR stands for Status System Feedback System responsibility) that can be used to delimit the impact of a certain usability feature in the design is as follows.

**SSF_SR-1**: Be aware of system statuses and their changes.

**Generic component responsibility**: Certain Domain Components can execute actions that will change one or more application statuses. A StatusManager Component is responsible for monitoring said Domain Components and listens for their status-altering actions. A Status Component is responsible for holding all the information relating to a particular status and for modifying it according to StatusManager orders (please see SSF_SR-2 and SSF_SR-3 for details on how this happens). All Status Components can have one active status value at any given time (i.e. “online status” can be ‘online’, ‘idle’, ‘busy’, ‘offline’, etc.). The component responsible for handling user events (UI) must monitor all Status Components and notify the user of any changes.

**SSF_SR-2**: Handle user-initiated status changes.

**Generic component responsibility**: The component responsible for handling user events (UI) listen for user actions and order their execution. The component in charge of delegating actions (if any) is responsible for ordering the appropriate Domain Component to execute said action. Upon execution of actions that are sta-
tus-changing, each Domain Component is responsible for notifying any interested parties (specifically the Status Manager Component, in this case). The StatusManager component then forwards the updated information onto the appropriate Status Component. Said Status Component is then responsible for determining the effect, if any, that the received information will have on its current active status value. It will, when applicable, change said value and notify any interested parties (specifically the UI Component in this case). The UI Component will update the status display for every notification of status change received.

**SSF_SR-3:** Handled system-initiated status changes.

**Generic component responsibility:** Upon execution of actions that are status-changing—invoked by any other class in the system or an external source—each Domain Component is responsible for notifying any interested parties (specifically the Status Manager Component, in this case), as is the case when such an action is invoked by the user through the UI (See SSF_SR-2). The StatusManager component then forwards the updated information onto the appropriate Status Component. Said Status Component is then responsible for determining the effect, if any, that the received information will have on its current active status value. It will, when applicable, change said value and notify any interested parties (specifically the UI Component in this case). The UI Component will update the status display for every notification of status change received.

**SSF_SR-4:** Present system status notifications to users.

**Generic component responsibility:** The UI Component is responsible for knowing how and where each status (and its possible values) are displayed within the interface, and thus update it accordingly upon reception of notifications of status value change.

### 4.2 SSF architectural component responsibilities

In this section we describe how the SSF impacts in the design of the software architecture, for which we have selected a layered architecture as this is one of the most common architectural styles used in many modern software applications. As the level of responsibility of each layer in the architecture is different, we need to map the system responsibility for each use case of the SSF mechanism. The generic responsibilities of SSF mechanism at the design level are as follows:

**Display status:** This component is responsible to show the changes of the status motivated by a user request or a change in the system status, and inform users about the new status of the system or as a result of a specific action.

**Status component:** A status component supports all the information related to one system status and notifies the subscribers of any change in a particular status.
**Status manager component:** This component handles the management of the status changes based on user and system requests and which to which new status the system must change and which information must be updated.

**Domain component:** A Domain Component represents the part of the system that is ultimately responsible for executing the actions requested by the user. For instance, in an email program, by clicking the ‘Send’ button may have many intermediate effects (checking that the Subject field is not empty, loading attachments, etc), but the part of the system that is actually responsible for sending the email would be referred to as the Domain Component in the context of these mechanisms.

More specifically, we detail the concrete responsibilities of SSF for a layered architecture. The *display status* component is supported by the presentation layer of any client-server Web or mobile application, as the status display shows the results of the new status to the clients at the presentation layer. The *status manager* and *status* component act at the application processing layer, where the functionality of the application, accepting user and system requests, changes the status accordingly to a new one and inform the subscribers about a new status. Finally, the *Domain component* is represented by the *DomainClass*, and it must be considered an entity to be substituted at design time by the class that actually performs the requested task. As a result, Table 1 shows how the responsibility of each use case affects to each component for the layered architecture. However, because the data management layer do not handle any responsibility in the use cases mentioned before and do not support any component regarding the SSF mechanism, so we didn’t represent it the table.

<table>
<thead>
<tr>
<th>System Responsibility</th>
<th>Presentation Layer</th>
<th>Application Processing Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSF_SR-1 ( \text{Be aware of system statuses (and their changes)} )</td>
<td>The <em>Presentation Layer</em> must subscribe to each <em>Status</em> object upon system initialization.</td>
<td>The <em>Status</em> object holds all the information related to one system status and the means to change and query this information. It must notify all subscribers (<em>Presentation Layer</em>) of any changes.</td>
</tr>
<tr>
<td>Display Status</td>
<td>Status Manager</td>
<td>Status</td>
</tr>
<tr>
<td>Status Manager</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 1. USABILITY-ENABLING DESIGN GUIDELINE: CONCRETE OBJECT RESPONSIBILITIES (3-layered architecture)**
### Table 1: Usability-Enabling Design Guideline: Concrete Object Responsibilities (3-Layered Architecture)

<table>
<thead>
<tr>
<th>System Responsibility</th>
<th>Objects and Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Presentation Layer</strong></td>
<td><strong>Application Processing Layer</strong></td>
</tr>
<tr>
<td><strong>Display Status</strong></td>
<td><strong>Status Manager</strong></td>
</tr>
<tr>
<td>SSF_SR-2 Handle user-initiated status changes</td>
<td></td>
</tr>
<tr>
<td>The Presentation Layer listens to user's requests for execution actions and forwards it to the Application Processing layer. The Presentation Layer displays the status display for every notification of status change received.</td>
<td>The StatusManager determines the corresponding Status object to update and does so with the information sent forth by the DomainClass</td>
</tr>
<tr>
<td>SSF_SR-3 Handle system-initiated status changes</td>
<td></td>
</tr>
<tr>
<td>The Presentation Layer changes the status display for every notification of status change received.</td>
<td>The StatusManager determines the corresponding Status object to update and does so with the information sent forth by the DomainClasses</td>
</tr>
<tr>
<td>SSF_SR-4 Present system status notifications to users</td>
<td></td>
</tr>
<tr>
<td>The Presentation layer knows which type of status notification to give for each status change. It also knows how and where to display each type of status notification and does so upon notification of Status objects.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2 describes the UML package diagram of a three-layered architecture representing the distribution of the classes belonging to the SSF mechanism. As we can see, these classes should be attached to those describing the functionality.
of a concrete application and which new relationships must be added to incorpo-
rate the SSF mechanism in the software architecture.

Figure 2: Enabling System Status Feedback usability mechanism in a UML
package diagram

5. User Preferences

The User Preferences usability mechanism covers the user need in a centralized
place or database where the user can alter the application’s settings. Different
types of user preferences can be organized around categories and let user to tailor
these or set their personal preferences for a given time frame. Moreover, in case of
any software problem, mobile phone users can roll back these changes and set the
default settings stored in the mobile phone. The use cases belonging to this mechanism are described in Figure 3 and are the following:

**PREF_UC-1 SavePreferencesForUser**: Upon making changes to one or more preferences, the user requests for them to be saved. This will trigger the included use case **PREF_UC-4 StorePreferenceValuesToPersistence**.

**PREF_UC-2 LoadCannedSettings**: The user requests a group of canned settings to be loaded, allowing him to set a number of preferences all at once. This use case triggers **PREF_UC-5 LoadPreferenceValuesFromPersistence** when loading the canned setting to use.

**PREF_UC-3 LoadPreferencesForUser**: The user requests his preferences to be loaded, directly or indirectly. For example, starting the application is an indirect way to request preferences to be loaded in certain systems. This use case also triggers **PREF_UC-5 LoadPreferenceValuesFromPersistence**, as each of the preferences to load need to be ‘filled in’ with their value, stored in persistence.

**PREF_UC-4 StorePreferenceValuesToPersistence**: This use case is triggered by **PREF_UC-1 SavePreferencesForUser** every time a user chooses to save his current preferences. It writes the preference values onto the predetermined physical medium.

**PREF_UC-5 LoadPreferenceValuesFromPersistence**: This use case is triggered by **PREF_UC-2 LoadCannedSettings** when loading a canned setting. Each of the default values for the preferences contained in that setting need to be loaded from persistence.

![Figure 3: Preferences Use Case Model](image)
Like in the SSF usability mechanism, the description of the uses cases for Preferences including the generic component responsibilities are shown in Table 2.

5.1 User Preferences generic component responsibilities

Based on the use cases of Figure 3, the major usability design guidelines and the generic component responsibilities for the User Preferences usability mechanism is as follows (PREF_SR stands for User Preferences System responsibility).

PREF_SR-1: Set preferences.
Generic component responsibility: A Preference component holds the information related to a single ‘live’ system preference, minimally: its name (i.e. Background color), their possible values (i.e. green, red, blue) and the current active value it may have. A preference is ‘live’ once it’s been loaded (as opposed to a preference setting that may be stored in the hard drive for later use). A PreferencesManager component is responsible for knowing, handling and retrieving all live Preference components within the system. A Setting Component represents a group of predetermined value pairs (preference name – preference value) that can be loaded from the hard drive (through the StorageFront Component) and rolled out into the live preferences. The StorageFront component represents the link between the application logic and the repository where the preference values are saved. A Setting will load its values through the StorageFront, as only this class has direct access to the information stored in the hard drive (or other media).

PREF_SR-2: Provide default values.
Generic component responsibility: The Preference component is also responsible for knowing what (if any) is its default value and for setting itself to that value if/when requested by the UI.

PREF_SR-3: Allow 'canned settings'.
Generic component responsibility: A SettingsManager is responsible for loading stored Settings when asked by the UI.

PREF_SR-4: Organize preferences.
Generic component responsibility: If preferences are to be grouped, a Group Component is responsible for holding related preferences and for providing the UI with them.
5.2 User Preferences architectural component responsibilities

In this section we describe the responsibilities of the major components of the User Preferences usability mechanism and how their role in a three-layered software architecture.

Preference component: This component is responsible for holding the basic data related to a ‘live’ preference where an attribute is set to a value (e.g.: the wireless network provider is set to a certain ‘local operator’).

Settings component: The Setting component represents a group of Preferences with an assigned value (e.g.: using canned settings).

PreferenceManager component. This component handles individual Preferences within the system.

StorageFront component: The StorageFront Component stores and retrieves preference values into persistence. It can access the physical media where these values are stored.

SettingsManager component: This component is in charge for saving and loading Settings upon a request to the system.

User component: The User Component is in charge for holding and accessing a sole Settings component. This particular Settings component holds all the preference values stored for this particular User.

Group component: The Group component handles one or several Preference objects, where preferences are often grouped in a tree structure.

Each of the aforementioned components can be represented as classes in the architecture with the following functionality. The Preference Component is represented by the Preference class and it handles the currently assigned (or active) value that the user has set or loaded. Preference objects are always contained within a Setting component, which is represented by the Setting class in the architecture. In addition, the so-called ‘canned settings’ are described, for example, by a single Setting object containing a certain number of preferences with an assigned value. In multi-user systems, each User will contain a single Setting object, holding and managing all of its preferences at any time.

The PreferencesManager Component is represented by the PreferencesManager class, and it is responsible for ordering the Preferences that change and to retrieve these when they are requested. The SettingsManager Component represented by the SettingsManager class is the responsible for ordering newly created settings to be saved, and to retrieve them. The User Component’s responsibilities are carried out by the User class, in charge of holding and managing the Setting object that contains all users’ preferences. Moreover, the Group Component is represented by the Group class for organizing and arranging Preferences according to a particular structure. Finally, the StorageFront Component is represented by the StorageFront parent class, and by any of its subclasses. These classes are responsible for storing any saving data on a physical medium. Each subclass of
**StorageFront** implements this functionality for each needed particular physical medium (e.g.: a Database or a text file). All these classes representing low-level component responsibilities are their role for usability in a layered architecture are described Table 2 and shown in Figure 4. However, in order to simplify the description of Table 2, we have grouped some the classes described before in more than one column (e.g.: The PreferencesManager class includes the functionality of Preference and Group classes).

<table>
<thead>
<tr>
<th>System Responsibility</th>
<th>Objects and Layers</th>
<th>Presentation Layer</th>
<th>Application Processing Layer</th>
<th>Data Mgmt. Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PREF_SR-1</strong></td>
<td></td>
<td>User</td>
<td>Preferences Manager</td>
<td>Settings Manager</td>
</tr>
<tr>
<td>Set Preferences</td>
<td></td>
<td>The User requests its Setting object to update all of the sent preference values. Requests are forwarded to the Preferences Manager</td>
<td>The Preferences Manager orders each Preference to update itself with the new value. Each Preference object updates its value to be the new value sent. The Preference object updates itself as requested by the Preferences Manager</td>
<td>For each value sent, the Setting object within the User updates its internal Preference objects. The Setting saves itself once all of its internal Preferences have been updated. The Setting object rolls itself out by loading all of its pref-value pairs from the StorageFront and loading them onto the ‘live’ Preferences, via the PreferencesManager. For each pref name in the received pref-value pairs, the Setting asks the PreferencesManager to update the corresponding Preference object with the designated value</td>
</tr>
<tr>
<td>System Responsibility</td>
<td>Objects and Layers</td>
<td>Application Processing Layer</td>
<td>Data Mgmt. Layer</td>
<td>StorageFront</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>-----------------------------</td>
<td>------------------</td>
<td>--------------</td>
</tr>
<tr>
<td><strong>PREF SR-2</strong>&lt;br&gt;Provide Default values</td>
<td>The User forwards the call to its Setting object</td>
<td>The Preference object, responsible for knowing and setting its default value, sets its currentValue to that default</td>
<td>The Setting orders each Preference to reset itself to its defaultValue</td>
<td>The Storage Front writes the updated Setting to the appropriate storage medium.</td>
</tr>
<tr>
<td><strong>PREF SR-3</strong>&lt;br&gt;Allow ‘canned settings’</td>
<td>The User loads the canned setting via the Settings Manager object</td>
<td>The Preferences Manager orders each Preference to update itself with the new value. The Preference object updates itself as requested by the Preferences Manager</td>
<td>The Setting rolls itself out by loading all of its pref-value pairs from the StorageFront and loading them onto the ‘live’ Preferences, via the PreferencesManager. For each pref name in the received pref-value pairs, the Setting asks the PreferencesManager to update the corresponding Preference object (live) with the designated value.</td>
<td>The StorageFront loads all of the pref-value pairs belonging to the Setting in question and returns them to it.</td>
</tr>
<tr>
<td><strong>PREF SR-4</strong>&lt;br&gt;Organize Preferences</td>
<td>When preferences are loaded these are displayed in groups or trees if applicable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4: Enabling User Preferences usability mechanism in a UML package diagram
6. A Mobile Traffic Complaint System

In order to study the effects of usability in the software architecture of a real application, we describe in this section how we introduced the aforementioned usability mechanisms in a mobile application for smart phones. The system consists of a modern Android application for managing the traffic tickets (M-Ticket) of the local police. The mobile application captures the data of traffic infractions and sends them to a Web server. The M-ticket application enables the policeman to capture an image and the location of the infraction via GPS and introduce the data of the car and the infraction using a text form. In this chapter we will not care about the functionality of the Web application located in the police station to manage the traffic tickets received from the mobile application, as we will only focus on the usability of the mobile app.

6.1 Usability requirements

The M-ticket system was developed in 2010, but it only included usability features regarding the design of the user interface, colours, size of menu buttons, and navigation. Because none of the usability mechanisms described in Section 3 was included, between 2012 and 2013 we decided to add and the two mechanisms described in this chapter and evaluate its impact on the architecture. Consequently, the first step was to define the following usability requirement aimed to support the introduce usability mechanisms in the M-ticket application, such as we describe below.

**UR1. Alert message to users for pending tickets:** In case of a loss of the radio signal between the mobile terminal and the server, an alert message will be displayed to the user of the M-ticket application indicating that the ticket cannot be sent. In addition, the mobile M-ticket application will show in the status bar an icon advertising the user that there are pending tickets to be sent.

**UR2. Configure alert messages:** The user can configure in the application the following options for the alert messages: activate/deactivate shaking the phone when a new alert arise, activate/deactivate a sound, and the number of times an alert should be notified to the user.

Both usability requirements must be implemented in terms of concrete usability mechanisms, such as those described in this chapter.
6.2 Impact on the software architecture

This section describes the changes we performed over the software architecture of the M-Ticket system to support the usability requirements and how these impacted on the existing functionality. Figure 5 shows the new package diagram of the modified software architecture. The three layers of the design are as follows:

(a) the presentation layer containing the entry screen to the Android applications,
(b) the business logic layer of the M-ticket application contains the functionality of the app and the usability mechanisms introduced, and (c) the middleware and data access layer supporting the connection to the GPS position system and images captured by the phone which are sent to the Web server database.

The three-layered architecture of the M-Ticket application depicts the new classes (red colour) introduced in the design and the classes that changed (blue colour) and handles the two usability mechanisms (i.e.: System Status Feedback and User Preferences) introduced in the system, such as we explain below:

**System Status Feedback:** As we can see in Figure 5, in the Android application we modified two of the existing classes (NewComplaint, SendComplaint) in application logic tier in order to support the system status feedback mechanism. The NewComplaint class allows the policeman to create a new traffic ticket using the Complaint class shown in Figure 5. The functionality of the M-ticket app implements also the location of the vehicle using the GPS of the mobile phone and it sends the form, the location, and a picture of the vehicle to a remote server. The changes introduced by the “Status Feedback” mechanism affect to the notifications sent to the policeman using the mobile phone. In this way, we used the android.app.NotificationManager Android library to warn the user about the events that may happen, such as a complaint already sent, no GPS signal. In addition, the changes to the SendComplaint class refer to the list of pending complaints stored in the mobile phone before they are sent to the server. In those cases of no pending complaints stored in the phone the notifications will be removed.

**User Preferences:** Changing the user preference of the alert messages supported by the status feedback mechanism led us to introduce new classes (classes in red color in Figure 5) in the architecture. However, this usability mechanism affected to the functionality of all layers in the architecture such as we describe. In the presentation layer, the MainScreen class (classes in blue color in Figure 5) which acts as entry point of the mobile application once the user has logged onto the system, was modified to incorporate specific methods to set and edit the preferences of the alert messages. In the logic tier, we added two new classes, PreferenceManager and AlertMessages, which handle the specific preferences (i.e.: shake, sound, and repetition) of each alert message. Finally, the implementation of the classes supporting this usability mechanism require a new class, StorageFront located in the data access layer to store the user preferences. As we can see, there is another class in that layer, M-ticket An-
droid database, which represent where the user preferences are stored. We added this class for the sake of clarity for designers but in our system the storage of the user preferences data are located in a specific database of the M-ticket application.

Figure 5: Modified software architecture of the M-ticket application including the classes for the two usability mechanisms
In order to provide a better understanding of the classes we added and changed in the original architecture of the M-ticket app when usability was introduced, we describe in tables 3 and 4 the association between the generic components of each usability mechanism and the classes that implement such functionality in our system accordingly to the architecture of Figure 5.

Table 3. Mapping between the classes of the SSF usability mechanism and those implemented in the architecture of the M-ticket application

<table>
<thead>
<tr>
<th>Usability Mechanism</th>
<th>Generic component</th>
<th>Classes in the M-ticket architecture</th>
<th>M-ticket Architectural Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Status Feedback</td>
<td>Display Status</td>
<td>New Complaint</td>
<td>This class display the status to the user</td>
</tr>
<tr>
<td>Status Manager</td>
<td>N/A</td>
<td></td>
<td>We don’t need this functionality as we only support one type of status</td>
</tr>
<tr>
<td>Status Concrete Status</td>
<td>New Complaint</td>
<td></td>
<td>This class checks if there are pending complaints stored in the phone at updates the status when the complaints are sent to the server</td>
</tr>
<tr>
<td>Domain</td>
<td>NotificationManager (Android)</td>
<td></td>
<td>This library performs low level operations when the status changes and it assigns a ID for the status. In case of a loss in the connection between the phone and the server, the New Complaint class will inform the Notification Manager with an ID, which will be used by the Send Complaint class when the notification need to be removed</td>
</tr>
</tbody>
</table>
Table 4. Mapping between the classes of the User Preferences usability mechanism and those implemented in the architecture of the M-ticket application

<table>
<thead>
<tr>
<th>Usability Mechanism</th>
<th>Generic component</th>
<th>Classes in the M-ticket architecture</th>
<th>M-ticket Architectural Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Preferences</td>
<td>User</td>
<td>Main Screen</td>
<td>Users can configure the options of their alert messages using the Main Screen interface</td>
</tr>
<tr>
<td></td>
<td>Preferences Manager</td>
<td>Preferences Manager</td>
<td>It handles the preferences set by the user</td>
</tr>
<tr>
<td></td>
<td>Preference</td>
<td>Alert Messages</td>
<td>The alert message is the configurable preference supported by M-ticket</td>
</tr>
<tr>
<td>Group</td>
<td>N/A</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>Settings Manager</td>
<td>N/A</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>Setting</td>
<td>N/A</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>Storage Front</td>
<td>Storage Front</td>
<td>It acts as an intermediate class to store the preferences edited by the user</td>
<td></td>
</tr>
<tr>
<td>Mobile Phone Database</td>
<td>M-ticket database</td>
<td>This class relates the Storage Front with the access to the M-ticket database where preferences are stored</td>
<td></td>
</tr>
</tbody>
</table>

The mappings between the generic components of each usability mechanism and the concrete classes in the M-ticket application described in tables 4 and 5 guide software designers to introduce the concrete architectural responsibilities. Hence, software architects can use these mappings to determine the concrete responsibilities of new and existing classes in their application for supporting a particular usability mechanism. In our example, only one class is assigned to one component but in more complex mechanisms, several classes can be assigned to the same component. However, we do not suggest guidelines for coding the usability mechanisms as these may depend on the current functionality of the application and the code in which the usability feature will be added.
6.3 Usability and interactions between entities

As introducing usability in the software architecture induces new relationships between the existing classes from the old design and the new classes supporting the functionality of both usability mechanisms, new interactions between the entities participating in the possible M-ticket scenarios arise. Because other stakeholders might be interested in the interactions that happen when a usability mechanism is activated or invoked by the system or the user, we describe in Figure 6 an example of a sequence diagram which exemplifies the dynamicity of the system and the calls made between the mobile user, the M-ticket application, and the server.

![Sequence Diagram]

Figure 6: Interactions of the major entities in the M-ticket application entangled with the interaction induce by the SSF usability mechanism.
The aforementioned scenario shown in Figure 6 describes the interactions between the entities when a policeman sends a traffic ticket to the server application using the mobile device and the case where the usability mechanism SSF notifies the user if the ticket has been sent or the case when due to a loss of the radio connection the pending tickets have to be stored in the mobile device and the connection has to be checked again before the tickets are re-sent.

7. Discussion

This chapter illustrates the importance of usability mechanisms and their implications at the architecture level when usability requirements demand specific usability mechanism, such as those described in this work. The research question RQ1 is answered in sections 4 and 5, where we describe the architectural responsibilities for the two usability mechanisms analyzed. Also, we have highlighted this importance by describing the role and the use cases of both usability mechanisms and which are the generic and concrete component architectural responsibilities for different scenarios according to the use cases.

Research question RQ2 is partially answered in sections 4 and 5 but detailed on the example presented in section 6 were we detail how the usability mechanisms describe affect the architecture of a mobile application. Our experience discusses the implications at the design level of the changes that should be made for each of the usability mechanisms discussed in this chapter, and provides a guiding example that shows the mappings between the generic classes of each usability mechanism and real classes of the mobile application used.

In addition, the usability requirements UR1 and UR2 defined for the M-ticket application are handled by the two usability mechanisms (SSF and User Preferences) described in the chapter. As a result, 3 of the existing classes were modified and 4 new classes were added. In order to estimate the impact of introducing both usability mechanisms in the design, the classes modified in the architecture represent around the 19% of the overall design, while the new classes represent around the 20% of the existing design. We didn’t take into account the changes or new relationships between classes, but the overall design effort in terms of classes added or modified represent the 39% of the design. However, the implications at the code level might not be exactly the same, as for instance, implementing the system status feedback in the M-ticket application required only a 4.2% of new code. Hence, even if software designers perceive that redesigning the architecture to support a certain number of usability mechanisms involves changing or adding a considerable number of classes, the changes at the code level can be lower.

In the example shown we have only addressed two usability mechanisms but more usability features can be incorporated. However, dealing with them will have its corresponding impact and cost in a clear competition with other quality attributes. Therefore a trade-off between usability and its implications need to be care-
fully done for each application, identifying the usability requirements more relevant from the user perspective in each situation. This usability study should be done as soon as possible during the development process to avoid rework.

Additional trade-off analysis between usability and other quality requirements can be carried out in order to address the conflicts between the quality attributes. For instance, the results reported in [9] show that usability and reliability have a positive influence while usability and efficiency exhibit a negative influence. In our M-ticket application, we can say that the usability mechanisms introduced enhance the reliability of the application as mobile users are informed when a traffic ticket cannot be sent to the server. Hence, a quality trade-off analysis aimed to balance quality attributes becomes key relevant for software and product operation.

8. Conclusions

The introduction of usability mechanisms in any software has clear implication for the design. In this chapter we show how the architecture a mobile application is affected when we add usability mechanisms, and which classes are needed to handle the responsibilities of each usability mechanism. We have also outlined the role of the major classes in the architecture when these usability mechanisms need to be introduced in a mobile any application. In order to assess software designers about the introduction of usability mechanisms in their systems and based on our personal experience, we have described the case of a real mobile application for which two usability mechanisms were introduced.

Consequently, we provide evidence about the need and impact for considering usability requirements in mobile software development. The usability requirements we have referred in this work are general recommendations from the HCI community aimed to improve the usability of any software system. In this sense, our approach is complementary to existing works that address usability testing in similar applications.

Regarding the generalizability of the results, we observed from our study that many of the usability mechanisms analyzed in this work can be found and used in many Web applications or in mixed Web-mobile software. More specifically for mobile apps, as many modern smart-phones share a similar interface, we believe using the same usability mechanisms in other mobile apps will be the same (also for Tablet PC).
9. References


