Reconciling usability and interactive system architecture using patterns

Ahmed Seffah *, Taleb Mohamed, Halima Habieb-Mammar, Alain Abran

Human-Centered Software Engineering Group, Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada

1. Introduction

Software architecture is defined as the fundamental design organizations of a system; they are embodied in its components, their relationships to each other and the environment, and the principles governing its design, development and evolution [ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems]. In addition, it encapsulates the fundamental entities and properties of the application that generally insure the quality of application (Kazman et al., 2000).

In the field of interactive systems engineering, architectures of the 1980s and 1990s such as MVC (Goldberg, 1984) and PAC (Coutaz, 1987, 1990) are based on the principle of separating the functionality from the user interface. The functionality is what the software actually does and what information it processes. The user interface defines how this functionality is presented to end-users and how the users interact with it. The underlying assumption is that usability, the ultimate quality factor, is primarily a property of the user interface. Therefore separating the user interface from the application’s logic makes it easy to modify, adapt or customize the interface after user testing. Unfortunately, this assumption does not ensure the usability of the system as a whole.

We now realize that system features can have an impact on the usability of the system, even if they are logically independent from the user interface and not necessarily visible to the user. Bass observed that even if the presentation of a system is well designed, the usability of a system could be greatly compromised if the underlying architecture and designs do not have the proper provisions for user concerns (Bass et al., 2001; Raskin, 2000). We propose that software architecture should define not only the technical interactions needed to develop and implement a product, but also interactions with the users.

At the core of this vision is that invisible components can affect usability. By invisible components, we mean any software entity or architectural attribute that does not have visible cues on the presentation layer. They can be an operation, data, or a structural attribute of the software. Examples of such phenomena are commonplace in database modeling. Queries that were not anticipated by the modeler, or that turn out to be more frequent than expected, can take forever to complete because the logical data model (or even the physical data model) is inappropriate. Client-server and distributed computer architectures are also particularly prone to usability problems stemming from their “invisible” components.

Designers of distributed applications with Web interfaces are often faced with these concerns: They must carefully weigh what part of the application logic will reside on the client side and what...
part will be on the server side in order to achieve an appropriate level of usability. User feedback information, such as application status and error messages, must be carefully designed and exchanged on the client and server part of the application, anticipating response time of each component, error conditions and exception handling, and the variability of the computing environment. Sometimes, the Web user interface becomes crippled by the constraints imposed by these invisible components because the appropriate style of interactions is too difficult to implement.

Like other authors (Bass et al., 2001; Folmer and Bosch, 2003), we argue that both software developers implementing the systems features and usability engineers in charge of designing the user interfaces should be aware of the importance of this intimate relationship between features and the user interfaces. This relationship can inform architecture design for usability. With the help of patterns, this relationship can help integrate usability concerns in software engineering. Beyond proposing a list of patterns to solve specific problems, our long-term goal is to define a framework for studying and integrating usability concerns in interactive software architecture via patterns.

The second section discusses the related word dealing with the architectures for interactive software. In Sections 3 and 4, we focus on specific ways in which internal software properties can have an impact on usability criteria. In Section 5, we attempt to provide a more general, theoretical framework for the relationships between usability and invisible software attributes. Finally, we conclude with the future investigations.

2. Related work

A large number of architectures for interactive software have been proposed, e.g., Seeheim model, model-view-controller (MVC), arch/slinky, presentation abstraction control (PAC), PAC-amadeus and model-view-presenter (MVP) (Bass et al., 1998). Most of these architectures distinguish three main components: (1) abstraction or model, (2) control or dialog and (3) presentation. The model contains the functionality of the software. The view provides graphical user interface (GUI) components for a model. It gets the values that it displays by querying the model of which it is a view. A model can have several views. When a user manipulates a view of a model, the view informs a controller of the desired change. Fig. 1 summarizes the role of each these three components for an MVC-based application.

The motivation behind these architecture models is to improve, among others, the adaptability, portability, complexity handling and separation of concerns of interactive software. However, even if the principle of separating interactive software in components has its design merits, it can be the source of serious adaptability and usability problems in software that provides fast, frequent and intensive semantic feedback. The communication between the view and the model makes the software system highly coupled and complex.

The major weakness of this architecture is the lack provisions for integrating usability in the design of the model or abstraction components.

Len Bass and his colleagues (Bass et al., 2001) identified specific connections between aspects of usability (such as the ability to “undo”) and the model response (processed by an event handler routine). Their attention was limited to single-user desktop systems only and the scenarios need to be validated in practice.

Folmer and Bosch (Folmer and Bosch, 2003) discussed a usability framework which consists of three levels: problem domain, solution domain and the usability properties level. This framework expresses the relationship between design methods that allow for design for usability at the architectural level and the evaluation tools that allow assessment of architectures for the support of usability. This research needs case studies to determine its validity and consider other application domains rather than e-commerce software.

To study these intimate relationships between the model and the interface, we proposed the following methodological framework to:

1. Identify and categorize typical design scenarios that illustrate how invisible components and their intrinsic quality properties might affect the usability.
2. Model each scenario in terms of a cause/effect relationship between (a) the attributes that quantify the quality of an invisible software entity and (b) well-known usability factors such as efficiency, satisfaction, etc.,
3. Suggest new design patterns or improve existing ones that can solve the problem described in similar scenarios.
4. Illustrate, as part of the pattern documentation, how these patterns can be applied within existing architectural models such as MVC.

3. Identifying and categorizing typical scenarios

The first step in our approach for achieving usability via software architecture and patterns is to identify typical situations that illustrate how invisible components of the model might affect usability. Each typical situation is documented using a scenario. Scenarios are widely used in HCI and software engineering (Carroll, 2000). Scenarios can improve communication between user interface specialists and software engineers who design invisible components – this communication is essential in our approach. Within our approach, we define a scenario as a narrative story written in natural language that describes a usability problem (effect) and that relates the source of this problem to an invisible software entity (cause). The scenario establishes the relationship between internal software attributes that are used to measure the quality of the invisible software entity and the external usability factors that we use for assessing the ease of use of the software systems.

The following are some typical scenarios we extracted from our day-to-day experiences and from a literature review. Other researchers also proposed other scenarios (see Kazman and Leonard, 2002). The goal of our research was not to build an exhaustive list of scenarios, but rather to propose a methodological framework for identifying such scenarios and to define patterns that be used by developers to solve such problems. The scenarios are therefore intended as illustrative examples.
3.1. Scenario 1: time-consuming functionalities

It is common for some of the underlying functionalities of an interactive system to be time-consuming. Several quality attributes can increase the time for executing these functionalities. A typical situation is the case where a professional movie designer expects high-speed Internet access when downloading large video files, but the technology of Internet connection makes configuration overly difficult.

The user needs feedback information to know whether or not an operation is still being performed and how much longer he will need to wait, but sometimes this information is not provided. Feedback tends to be overlooked; in particular, when the designers of the user interface and those developing the features are not the same and that there is a lack of communication between them. The feedback is important but how much feedback is needed? No feedback is bad but too much feedback can also be a negative influence on usability as it may overwhelm the user with information.

3.2. Scenario 2: updating the interface when the model changes its state

Usability guidelines recommend helping users understand a set of related data by allowing them to visualize the data from different points of view. A typical method is to provide graphical and textual representations of the same underlying data model.

Whenever the data model changes, the underlying model should update the graphical and the textual representations. In certain cases, the system might not be designed to automatically update all views when one view changes. This can result in inconsistent views that can in turn increase the user’s memory load, frustration, and errors.

3.3. Scenario 3: performing multiple functionalities using a single control

It is recommended to use “a dedicated control approach” for each functionality and in particular for critical functions, even at the expense of more buttons and menus. When “a single control approach” performs multiple operations, it requires a complex menu structure and choice of modes, which increases the likelihood of mode errors and other usability problems. Usually a more hybrid approach is taken where all the functionality is still available but not directly visible (e.g., drop down menus in word that only show the most commonly used tasks). Also the user has the ability to define macros and tie several tasks into one step.

Unfortunately, there is a design tradeoff between simplicity in appearance and simplicity in use. This is a dangerous design trap. Ales, consumers (and organizations) make purchase decisions based on appearance first; so this is a fundamental conflict (Norman, 2002).

3.4. Scenario 4: invisible entities keep the user informed

We know that providing the user with an unclear, ambiguous or inconsistent representation of the system’s modes and states can compromise the user’s ability to diagnose and correct failures, errors and hazards. This can happen when a system functionality allows the user to visualize information that competes or conflicts with previously displayed information in other views.

To avoid such situations, it is important for the functionality developers to communicate the system’s modes and states to the user interface designer. User interface designers should inform the developers about all the visible consequences related to the states and modes of the systems.

3.5. Scenario 5: providing error diagnostics when features crash

When a feature failure occurs due to, for example, exception handling, the interface sometimes provides unhelpful error diagnostics to the user.

The user should be notified of the state that the system is currently in and the level of urgency with which the user must act. The system feature should help the user to recognize potential hazards and return the system from a potentially hazardous state to a safe state. Messages should be provided in a constructive and correct manner that helps restore the system to a safe state.

3.6. Scenario 6: technical constraints on dynamic interface behavior

Particularly in Web-based transactional systems, technical and logistic constraints can severely limit dynamic behavior of the interface within a given page. It can therefore be difficult or impossible to design elements that automatically update as a result of an action elsewhere on the same page. For example, in a series of dependent drop-down lists “Country”, “Province” and “City”, it may be impossible to automatically update “Province” as a function of the “Country” selection.

These technical constraints against dynamism are often imposed in Web-based client-server contexts due to the dictum that the business rules must be separate from the user interface. Dynamic interface behavior can require the user interface to have a degree of intelligence that incorporates certain business rules, which conflicts with the “separate layers” dictum. The alternative is for the client to call the server more frequently to refresh the page dynamically, but architects tend to avoid this approach because of the presumed extra demand on bandwidth.

There is no easy solution to this problem. The most important principle in this situation is to analyze user needs relating to dynamism before making technology decisions that could have an impact on dynamism. Transactional systems often require considerable dynamism, whereas purely informational systems can often get by without dynamism in the user interface. If it is unacceptable for business rules to be incorporated into the client, then it might be possible to make a business case for increasing the network bandwidth so as to better support pseudo-dynamic behavior, involving more frequent page refreshes through calls to the server.

The preceding scenarios are used as an illustrative sample. In total, we have identified more than 24 scenarios. Len Bass also described a list of 26 scenarios, some of which were a source of inspiration for our work. Providing an exhaustive list of scenarios is certainly useful from the industry perspective. However, our goal for this research is to better understand and validate how software features affect usability in general, and as such, our focus is to model the scenarios in terms of a cause/effect relationship. This relationship connects the quality attributes of invisible components with recognized usability factors. Section 5 details this perspective.

4. Patterns as solutions to the problems documented as scenarios

There are different ways to document solutions for the problems described in the preceding scenarios. In our framework, we have been using design patterns (Alexander et al., 1997; Gamma et al., 1995). Since the relationship between usability and internal software properties define the problem, it has been added into the pattern descriptions that follow. This measurement relationship is what makes a pattern a cost effective solution. In short, if a pattern does not improve at least one of the factors described in the measurement relationship, then it is not a good pattern for the problem described in the scenario. From a cost effective point of view its
also very useful to identify such scenario’s & patterns during an early stage when its still cheap to change design decisions.

In this section, we present two different types of architecture-sensitive patterns:

- Software design patterns. The aim of these design patterns is to propose software designs and architectures for building portable, modifiable and extensible interactive systems. A classical pattern of this category is the Observer that acts as a broker between the user interface (views) and the model (Gamma et al., 1995). When the observers receive notification that the model has changed, they can update themselves. This pattern provides a basic solution to the problem described in scenario 3.

- Interaction design patterns, defined at the level of interaction (Wелиe, 1999). These are proven user experience patterns and solutions to common usability problems. A number of pattern languages have been developed over the last few years. Among them, the Common Ground and Amsterdam catalogues play a major role (Tidwell, 1997; Wелиe, 1999).

Software design patterns, widely used by software engineers, are a top-down design approach that organizes the internal structure of the software systems. Interaction design patterns, promoted by human computer interaction practitioners, are used as a bottom-up design approach for structuring the user interface. Our position is that these two categories of patterns need to be combined in order to providing an integrated design framework to problems described in our scenarios. To illustrate how these diverse patterns can be combined to provide comprehensive solutions, in the following sections we describe our six scenarios using interaction and design patterns.

A number of de facto standards have emerged to document patterns; the following is a proposal for the canonical pattern form to be used. Basically a variant of the Coplien form (Coplien, 2001):

- “Name” is a unique identifier.
- “Context” refers to a recurring set of situations in which the pattern applies.
- “Forces”: the notion of force generalizes the kinds of criteria that we use to justify designs and implementations. For example, in the study of functionality, the main force to be resolved is efficiency (time complexity). However, patterns deal with the larger, harder-to-measure and conflicting sets of goals and constraints encountered in the design of every component of the interactive system.
- “Problem” refers to a set of constraints and limitations to be overcome by the pattern solution.
- “Solution” refers to a canonical design form or design rule that someone can apply to resolve these problems.
- “Resulting context” is the resulting environment, situation, or interrelated conditions.
- “Effects of invisible components on usability” which defines the relationship between the software quality attributes and usability factors.

### 4.1. Software design patterns

The first pattern that we have considered is the Abstract Factory pattern, which provides an interface for creating families of related or dependent objects without specifying their concrete implementations (e.g. The Toolkit class). In other words, this pattern provides the basic infrastructure for decoupling the views and the models. Given a set of related abstract classes, the Abstract Factory pattern provides a way to create instances of those abstract classes from a matched set of concrete subclasses. The Abstract Factory pattern is useful for allowing a program to work with a variety of complex external entities such as different windowing systems with similar functionality. The second pattern we implemented within our framework is the Command pattern, which complements the abstract factory by reducing the view/controller coupling.

Another pattern that complements the three basic patterns mentioned in the last paragraph is the Working Data Visualization pattern.

**Name**: Working data visualization

**Scenarios addressed 2. Updating the interface when the model changes its state**

**Problems**

If the user cannot see working data in different view modes so as to get a better understanding of it, and if switching between views does not change the related manipulation command, then usability will be compromised.

**Context**

Sometime users want to visualize a large set of data using different point of view, so as to better understand what they are doing and what they need to edit to improve their documents.

**Forces**

- Users like to gain additional insight about working data while solving problems.
- Users like to see what they are doing from different viewpoints depending on the task and solution state.
- Different users prefer different viewpoints (modes).
- Each viewpoint (mode) should have related commands to manipulate data.

**Solution**

Data that is being viewed should be separate from the data view description, so that the same data can be viewed in different ways according to the different view descriptions. The user gets the data and commands according to the user-selected view description.

**Resulting context**

After applying theses patterns, the user is able to get a better understanding of interrelated data viewed in different screens. Others patterns for data visualization may apply to structure the information in cognitively acceptable chunks.

**Effects of invisible components on usability**

**Effect 1**

- Quality attributes of invisible components: integrity
- Usability factors affected: visual consistency

Many of these implementation details are shared by multi channel access pattern for working data visualization pattern (Folmer et al., 2006).

Other relevant patterns we used includes (2001), namely event handler, complete update, and multiple update (Sandu, 2001). We use them to notify and update views (scenario 2) using traditional design patterns such as observer and abstract factory. We incorporated these patterns into the sub-form pattern that groups the different views in the same container, called the form (Table 1). The event handler, complete update, and multiple update patterns can be applied in two phases. The first phase changes the states of the user interface models in response to end user events generated by the visual components, and the second phase updates the visual components to reflect the changes in the user interface model. Since the update phase immediately follows the handling phase, the user interface always reflects the latest changes.

The next example of software design patterns we propose is the reduce risk of errors pattern.
### Example of design patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event handler</td>
<td>How should an invisible component handle an event notification message from its observable visual components?</td>
<td>Create and register a handler method for each event from observable visual components</td>
</tr>
<tr>
<td>Complete update</td>
<td>How to implement behavior in the user interface to update the (observer) visual component from the model</td>
<td>Assume all (observer) visual components are out-of-date and update everything</td>
</tr>
<tr>
<td>Multiple update</td>
<td>How to implement changes in the model of sub-form to reflect parent of sub-form, child of sub-form, siblings of sub-form</td>
<td>Each sub-form should notify its parent when it changes the model. The parent should react to changes in the sub-form via the event handler and update its children components via Complete Update</td>
</tr>
<tr>
<td>Sub-form</td>
<td>How to design parts of user interfaces to operate on the model in a consistent manner</td>
<td>Groups the components that operate on the same model aspect into sub-forms</td>
</tr>
</tbody>
</table>

**Name**: Reduce risk of errors  
(Scenarios addressed 2. Updating the interface when the model changes its state; 4. Invisible entities keep the user informed)

**Problem**: How can we reduce the likelihood of accidents arising from hazardous states?

**Forces**
- Hazardous states exist for all safety-critical systems; it is often too complex and costly to find every hazardous state by modeling all system states and user tasks;
- Risk can be effectively reduced by reducing the consequence of error rather than its likelihood;
- When a hazardous state follows a non-hazardous state, it may be possible to return to a non-hazardous state by applying some kind of recovery operation.

**Solution**
Enable users to recover from hazardous actions they have performed. Recovering a task is similar to undoing it, but promises to return the system to a state that is essentially identical to the one prior to the incorrect action. This pattern may be useful for providing a recovery operation giving a fast, reliable mechanism to return to the initial state. Recovering a task undoes as much of the task as is necessary (and possible) to return the system to a safe state.

**Resulting context**
After applying this pattern, users will have more immediate feedback on the consequences of their actions, increasing the understandability of the interface and reducing errors; in addition, time and effort to accomplish a task will be reduced in certain cases.

**Effects of invisible components on usability**

**Effect 1**
- Quality attributes of invisible components: functionality
- Usability factors affected: understandability

**Effect 2**
- Quality attributes of invisible components: suitability
- Usability factors affected: operability

4.2. Interaction design (HCI) patterns

Many groups have devoted themselves to the development of pattern languages. Among the heterogeneous collections of patterns, “Common Ground” and “Amsterdam” play a major role in this field and wield significant influence [Tidwell, 1997; Welie, 1999]. We also adapted and use some these patterns.

The first basic HCI pattern that we used is the progress indicator pattern [Tidwell, 1997]. It provides a solution for the time-consuming features scenario (scenario 1).

**Name**: Progress indicator  
(Scenarios addressed 1. Time-consuming functionalities)

**Problem**: A time-consuming functionality is in progress, the result of which are of interest to the user. How can the artifact show its current state to the user, so that the user can best understand what is happening and act on that knowledge?

**Forces**
- The user wants to know how long they have to wait for the process to end.
- The user wants to know that progress is actually being made, and that the process has not just “hung.”
- The user wants to know how quickly progress is being made, especially if the speed varies.
- Sometimes it is impossible for the artifact to know how long the process is going to take.

**Solution**
Show the user a status display of some kind, indicating how far along the process is in real time. If the expected end time is known, or some other relevant quantity (such as the size of a
file being downloaded), then always show what proportion of the process has been finished so far, so the user can estimate how much time is left. If no quantities are known – just that the process may take a while – then simply show some indicator that the process is ongoing.

**Resulting context**

A user may expect to find a way to stop the process somewhere close to the progress indicator. It is almost as though, in the user’s mind, the progress indicator acts as a proxy for the process itself. If so, put a “stop” command near the Progress Indicator if possible.

**Effects of invisible components on usability**

**Effect 1**

– Quality attributes of invisible components: performance
– Usability factors affected: user satisfaction

The second pattern integrated in our framework is the keep the user focused pattern, which brings an integrated solution to the problems described in scenarios 2, 3 and 4.

**Name:** Keep the user focused

**Scenarios addressed** 2. Updating the interface when the model changes its state; 3. Performing multiple functionalities using a single control; 4. Invisible entities keep the user informed

**Context**

An application where several visual objects are manipulated, typically in drawing packages or browsing tools. Problem: How can the user quickly learn information about a specific object they see and possibly modify the object?

**Forces**

– Many objects/views can be visible but the user usually works on one object/view at a time.
– The user wants both an overview of the set of objects and details on attributes and available functions related to the object he or she is working on.
– The user may also want to apply a function to several objects/views.

**Solution**

Introduce a focus in the application. The focus always belongs to an object present in the interface. The object of focus on which the user is working determines the context of the available functionality. The focus must be visually shown to the user, for example, by changing its color or by drawing a rectangle around it. The user can change the focus by selecting another object. When an object has the focus, it becomes the target for all the functionality that is relevant for the object. Additionally, windows containing relevant functionality are activated when the focus changes. This reduces the number of actions needed to select the function and execute it for a specified object. The solution improves the performance and ease of recall.

**Resulting context**

The “keep the user focused” pattern complements the software design patterns in the following situations:

– Helping users anticipate the effects of their actions, so that errors are avoided before calling the underlying features;
– Helping users notice when they have made an error (provide feedback about actions and the state of the system);
– Providing time to recover from errors;
– Providing feedback once the recovery has taken place.

**Effects of invisible components on usability**

**Effect 2**

– Quality attributes of invisible components: functionality
– Usability factors affected: understandability

Effect 3

– Usability factors affected: operability
– Quality attributes of invisible components: suitability

There is not a one-to-one mapping between software design patterns and HCI patterns. The problems described in a specific scenario can require any number of HCI and software design patterns, and each pattern may be affected by a number of problems described in different scenarios. In our approach, we argue that using even a few patterns can be very valuable, even without an entire pattern language.

Our list of patterns is not intended to be exhaustive. We still are considering some of the existing patterns (Newman and Lamming, 1995; Buschmann et al., 1996). However, most of the existing patterns have not originally been proposed to cope with the problem we are addressing. We are therefore adapting them as we did with the ones we introduced in this section.

5. **Modeling cause-effect relationships between software elements and usability**

In Sections 3 and 4, we focused on specific ways in which internal software properties can have an impact on usability criteria. In this section, we attempt to provide a more general, theoretical framework for the relationships between usability and invisible software attributes. In particular, among the huge or potentially infinite number of ways that invisible components can affect usability, we wish to understand whether there are specific places where we are more likely to find these relationships or effects. We also wish to know whether there is any structure underlying these relationships, which would allow us to define a taxonomy of how usability issues arise from invisible components.

5.1. **Traditional model of relationship between invisible software elements and usability**

Usability is often thought of as a modular tree-shaped hierarchy of usability concepts, rooted in the level of GUI objects, and abstracting progressively up to low-level usability criteria or measures and then high-level usability factors. Fig. 2 illustrates this definition of usability and its relationship to parallel “towers” of other software attributes.

Table 2 provides more detailed information on the software quality factors and criteria referred to schematically in the right-hand branch of Fig. 2. (In principle, each quality factor would form a separate branch.) In our work, we have adopted the software quality model proposed by ISO 9126. Table 2 is an overview of the consolidated framework we have been using (Seffah et al.,
The details of this framework are outside the scope of this paper. The table shows the criteria for measuring usability as well as five other software quality factors including functionality, reliability, efficiency, maintainability, and portability. This measurement framework automatically inherits all the metrics and data that are normally used for quantifying a given factor. The framework helps us to determine the required metrics for (1) quantifying the quality factors of an invisible software entity, (2) quantifying the usability attributes and (3) defining the relationships between them.

### 5.2. Taxonomy of usability issues arising from invisible components

Relationships between software attributes of invisibles components and usability factors have two properties:

1. They are lateral relationships between the modules of usability and architecture.
2. They are hierarchical relationships between two or more levels of description, since usability properties are a higher-level abstraction based on architectural elements.

Thus to understand the relationship, we need an approach that takes into account both modularity and hierarchy. In “the architecture of complexity” (1962), Herbert Simon discusses “nearly decomposable systems”. In hierarchical systems, interactions can be divided into two general categories: those among subsystems, and those within subsystems. We can describe a system as being decomposable into its subsystems. However, as a more refined approximation, it is possible to speak of a system being “nearly decomposable”, meaning that there are in fact interactions between the subsystems, and that these interactions are weak but non-negligible.

Nearly decomposable systems have two properties:

1. Modularity: in the short-run, the behavior of each subsystem is approximately independent of the other subsystems;
2. Hierarchy (or aggregation): in the long run, the behavior of any one subsystem depends in only an aggregate way on the other subsystems.

These properties indicate that in reality, the traditional model of usability is an oversimplification. Although the usability subsystem is fundamentally modular from the architecture, Simon’s principle of nearly decomposable systems predicts that it is possible for usability properties to be affected to some degree by architectural properties. Fig. 3 illustrates an interpretation of this alternative model of usability.

In Fig. 3, a node (usability property) at any level of usability can potentially be influenced by nodes at any lower level of architecture, or conceivably even by combinations of several different levels of architecture. Fig. 3 is a first approximation. Simon’s second principle of near-decomposability states that subsystems depend in only an aggregate way on other subsystems.

This principle implies that if architecture has an effect on usability, it will tend to be in an aggregate way and therefore at a higher level of architecture, rather than through the effect of an individual low-level architectural component. We interpret this principle to mean that the effects of architecture on usability will tend to propagate from levels of architecture that are closer to the level of usability, rather than farther away.

Therefore, to refine the model, we will assume that the most likely relationships occur between usability properties and the immediately closest lower architectural level, and that more distant architectural levels have an exponentially decreasing probability of having an effect on usability. The revised model, based on this assumption, is illustrated in Fig. 4. This model reflects a more clearly recursive definition of usability.

Based on Simon’s principles of nearly decomposable systems, we can conclude that these types of relationships between architecture and usability are the exception to the rule, but frequent enough that they should not be neglected.
Table 3: Examples of relationships between invisible software entities and usability factors

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Quality attributes of invisible components</th>
<th>Usability factors affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Performance</td>
<td>User satisfaction</td>
</tr>
<tr>
<td>2.</td>
<td>Integrity</td>
<td>Visual consistency</td>
</tr>
<tr>
<td>3.</td>
<td>Functionality</td>
<td>Understandability</td>
</tr>
<tr>
<td>4.</td>
<td>Suitability</td>
<td>Operability</td>
</tr>
<tr>
<td>5.</td>
<td>Recoverability</td>
<td>Attractiveness</td>
</tr>
<tr>
<td>6.</td>
<td>Interoperability</td>
<td>Adaptability</td>
</tr>
</tbody>
</table>

5.3. Application

This measurement model provides a framework within which to explore these exceptional ways that architecture can affect usability, so as to work toward a more complete model of usability. The model is useful because it helps us know where to look for relationships between architecture and usability. Further progress will require detailing the hierarchies on both sides of the tree, and considering each possible relationship between nodes at proximate levels. Another goal will be to provide other heuristic principles to further narrow down the likely interrelationships between these two branches.

Table 3 provides examples of the specific types of relationships that occur in the scenarios described in Section 2. The second column refers to the invisible object’s properties and software qualities identified in the right-hand branch of Figs. 2–4, and the third column represents the usability properties identified in the left-hand branch of those figures.

For example, scenario 1 can be modeled as a relationship that connects the performance of the software feature with certain usability attributes such as user satisfaction. It can lead to the following requirement related to scenario 1: “to ensure an 80% level of satisfaction, the maximum acceptable response time of all the underlying related feature should not exceed 10 s; if not the user should be informed and a continuous feedback needs to be provided”.

6. Conclusion and future investigations

In this paper, we first identified specific scenarios that how invisible software components can have an effect on the usability of the interactive system. Then, we provided a list of patterns that solved the problems described in the scenarios. This research effort can benefit software architecture designers and developers, who can use our approach in two different ways. First, the scenarios can serve as a checklist to determine whether important usability features (external attributes) have been considered in the design of the features and the related UI components. Secondly, the patterns can help the designer to incorporate some of the usability concerns in the design.

More than defining a list of scenarios and patterns that describe the effects of invisible software attributes on software usability, our long-term objective is to build and validate a comprehensive framework for identifying scenarios. The goal of the framework is to define these patterns as a relationship between software quality factors and usability factors. In this paper, we have suggested different HCI and software design patterns as solutions to the problems described in these scenarios and in similar ones. Every pattern has a set of problems to be solved and a set of goals to be achieved.

We expect that as we gain a better understanding of the relationship between interaction design patterns and software architecture patterns, this knowledge will affect the evolution of standards in architecture design and GUI software libraries. In fact, this has already started. Increasingly, developers are making proper use of standard GUI libraries and respecting interface design guidelines in a way that considerably increases the usability of interactive applications. However, more can be done in this direction, and the approach we have outlined in this paper is an attempt to build a better and more systematic understanding of how usability and software architecture can be integrated.

References