Achieving Usability via Design Architecture and Patterns:

The Things We Implement that Affect Usability

Ashutosh Paul

A Thesis in the

Department of Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

June 2003

Your file Votre référence
ISBN: 0-612-83918-4
Our file Notre référence
ISBN: 0-612-83918-4

Abstract

Achieving Usability via Design Architecture and Patterns: The things we implement that affect usability

Ashutosh Paul

Current, software architectures for interactive systems assumed that usability is only important when designing the user interface component including the presentation and dialogs. Therefore, most of the architectural models such as MVC and PAC de-coupled the application semantic and the user interface, which was essential for achieving usability. However, this fallacious dichotomy does not address all the usability concerns. For example, semantic feedback leads tight coupling between the application and the user interface, because it has to be analyzed at both the application and the presentation levels. There are two approaches to cope with semantic feedback. We can either place semantic feedback in the presentation or in the application. The two approaches result in much communication between the model and the views. This thesis proposes a new architectural model, called MVCforUsability, for incorporating usability concerns in software architecture. In particular, we discuss the cause/effect relationships that exist between the internal attributes that we generally use to assess the software quality and the usability factors that we use to quantify the usability of the software system. Our investigations propose the following methodological approach:

1. Identify the internal software attributes that affect software usability, such as modularity, functionality, etc.
2. Define the relationship between these internal attributes and external usability factors in terms of typical scenarios.
3. Describe design patterns or improve existing ones for each typical scenario.
4. Discuss how these patterns can be used to improve the MVC model.

Acknowledgements

It is with great pleasure that I acknowledge the efforts of the number of people who have contributed to this thesis. First and foremost, I thank my supervisor Dr. Ahmed Seffah who has given many ideas and advice. I also wish to thank my younger brother Prakash Paul for many useful suggestions. Finally, I gratefully acknowledge the help I received during the preparation of this Thesis from my friends.

Table of Contents

List of Figures

List of Tables

1 Introduction

In the usability engineering community, little has been done on how to integrate usability analysis into the software architectures. The goal of usability analysis is to ensure a usable product. The earlier in the design process the usability analysis are carried out and integrated into design process, the more likely is that the product is usable in the end of the product development life cycle. In this work we have tried to integrate usability attributes early in the development process by considering cause/effect relationship between the internal attributes of the software system as 'causes' and usability attributes as 'effect'. The perspective of this work is to emphasize the problems and to find solutions for integrating usability in Human-Computer Interface architecture and design phase.

1.1 The Problem

Software architectures have long been a key research topic in human-computer interaction. Humans play an active and essential role in the operation of interactive software and user interface (UI) has become an essential part of many software systems.

According to the survey on user interface programming - almost half of the code is devoted to the user interface part of software system, during design and implementation about half of the time is spent on the user interface, and during maintenance about one third of the time is spent on the user interface [Myers, Rosson, 1992].

For this reason, most interactive system architectures are based on the assumption of separation between the functionality and the user interface. The functionality is what the software actually does and what information it processes. The user interface defines how this functionality is represented to end-users and how user input is pressed.

The interactive system architectures have been proven useful, but they also introduce problems with respect to the usability. These problems occurs in particular in 'highly' interactive systems, where user interface and application model are semantically dependent on each other: the more a software system is interactive, the more coupling between its functionality and user interface. So, only by changing UI we cannot increase usability of the system, which was thought in 1980s and early 1990s that usability was primarily a property of the presentation of information. So, if we don't integrate the usability attributes in design process the system will not be usable only by changing UI.

1.2 Usability of visible versus invisible components

Usability of software has two aspects: the UI usability, which we called visible usability, and interactive system functional usability, which we called invisible usability. Most of the research on Human-Computer Interaction has focused on the visible usability aspect such as look and feel, colors, error messages clarity, device behaviors, language, etc.

In this thesis, we will demonstrate that there are many invisible elements of an interactive system, which are not either seen by the user or user concerns, but they can affect usability of the software. We will also explain how these aspects can be incorporated in the software architecture at the design phase. The following examples explain how invisible elements affect usability of the software system.

(A) Response time affects the performance of the system

If user wants to download a file form the server; the downloading depends on many internal parameters such as network bandwidth, computer speed, system's API, etc. If there is any problem with any related internal parameters it will either take longer time and simply does not download the file from the server. Generally user is not aware about those problems, because those problems are simply related with internal mechanism of the system.

(B) Exceptional handling can affect the user satisfaction

Users may put wrong inputs and system should tolerate them. The system can crash due to exceptional inputs. We can say system is crushed because it does not have proper exception handling mechanism. Why the system is crushed, it is fully beyond user to understand because exception handling is the internal mechanism of the system. So, we should make the system more robust by providing an internal mechanism to handle the exceptional inputs. This will result in more user satisfaction.

(C) Data validation can affect the appropriateness of user feedback

Sometimes users are not getting expected outputs in their TEXT FIELD when they are pressing corresponding BUTTON. This is caused due to improper validation of the inputs and outputs of the pressed BUTTON which is not either visible to the users or users' concerns. It is the designer concerns to build the proper validation mechanism behind the BUTTON to get the proper outputs in term of given inputs.

Intuitively, any invisible component can affect the usability. For example, let us consider a TEXT FIELD, which is used to display a piece of information. When users press a button, TEXT FIELD will be filled out with the output of the button pressed. If designers/developers don't implement validation mechanism (e.g., IF-THEN statement) for validating the inputs and outputs, the TEXT FIELD can be filled out with garbage that will not be readable or difficult to understand. So, we can say the validation mechanism is an invisible component that affects usability.

So, we can say cause/effect relationship establishes a relation between internal attributes such as data validation and one or more external attributes such as efficiency, helpfulness, etc. It creates a situation, which we will call a scenario where invisible components can affect the usability of software system.

In this thesis, we define internal and external attributes as follows:

1- Internal attributes are invisible software components that are used to measure the quality of the software system and its behaviors. Some internal attributes

like size, effort, and cost is easier to measure. Others are more difficult like code complexity. Furthermore, internal attributes are functionality, modularity, reuse, redundancy, structuredness, module coupling and cohesiveness.

2- External attributes refer to the quality factors that we generally use to measure the usability. Examples of usability factors include efficiency, user satisfaction, and effectiveness. ISO 9126 provides an exhaustive list of usability factors.

The following table (Table 1) summarized the list of internal and external attributes that we considered in this thesis.

Table 1: Some internal and external attributes with their definitions

Attributes	Definition
Internal	
Familiarity mechanism	Creates metaphor in user mind and leverage human knowledge.
Natural Mapping	Creates a clear relationship between what the user wants to do and the mechanism for doing it.
Data and commands aggregation	Aggregation is a process to gather information into an object.
Distinct views with same functionality mechanism	Same functionality for different view of the same data.
Undo/cancel mechanism	Ability of recovery.
Recognize user error mechanism (spell checking)	Ability of prediction human error through perceptual and cognitive analysis.

Attribute	Definition
Internal	
Customizability mechanism	Ability of adaptation of user interface with user habits and environment.
Robustness mechanism (fault tolerant):	Ability of tolerance with exceptional inputs.
Display system status mechanism (feedback):	Ability to display system dialogs.
Support multiple-visualization mechanism	Ability to see objects in different shape and structure.
External	
Learnability	Allows users to begin work quickly.
Efficiency	Enables a high degree of productivity.
Error Minimization	Mistakes are infrequent, but easy to recover from.
Adapt the system (Adaptability)	Personalization of the system with user environment.
Comfortability	Enjoyable to work with.

1.3 Software Architecture

Software architecture represents a common vehicle for communication among a system's stakeholders, and is the arena in which conflicting goals and requirements are mediated. Architecture is the organizational structure of a system. Architecture is the first artifact in the software design process that can be analyzed to determine how well its non-functional quality attributes (architecture quality requirements) are being achieved, and it also serves as the project blueprint. Architecture is also a description of the relationships among components and connectors. Components are identified and assigned

responsibilities that client components interact with connector interface. Connector interface specifies communication, controls mechanisms, and supports all component interactions needed to accomplish system behavior.

The following are some of the definitions that have been proposed:

- IEEE Std. 610.12-1990: A system is a collection of components organized to accomplish a specific function or set of functions.

- Garlan and Perry, guest editorial to the *IEEE Transactions on Software Engineering*, April 1995: Software architecture is "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."

- Bass, Clements, and Kazman - Software Architecture in Practice, Addison-Wesley 1997: 'The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

1.4 Interactive System Architectures

We define an interactive system as a system, which provides the user interface and the support for human activity. The user interface has two sides - inputs and outputs. In input side user inputs information indicated by various conventions and controls; and in output side machine provides feedback and other assistance to the user in command specification, and provides various forms of information

portrayal. Human have many motors and other capabilities that could be the basis for input and command specifications; similarly user has full range of senses that could be targets for system outputs.

According to William M. Newman and Michael G. Lamming: An interactive system supports communication in both directions, from user to computer and back. It does this in a way that enables it to follow the pace and direction of the user's activity. The user takes actions such as pressing buttons, pointing with a 'mouse' or typing in the text. The system reacts accordingly, perhaps by displaying information, perhaps by activating machinery or performing some other useful service, perhaps just by waiting for the user's next action. All of this takes place via the system's user interface, the part of the system that provides access to the computer's internal resources.

The main characteristics of interactive systems are:

1. Input events are triggered by user through input devices.
2. Action by users must trigger an output provided by the system.
3. Input and output are interleaved.

Following three fundamental things occur regarding event in interactive system:

- An event occurs – for example: the phone rings; users press button to open up a file.

- The event is detected by the system – for example: Apu hears the phone; system's 'A' drive gets light.
- The system reacts to the event – for example: Apu answers the phone; system brings up the file structure of the 'A' drive.

Four fundamental issues in interactive system design has to be addressed:

- Identify the human activity that the proposed interactive system will support.
- Identify the users who will perform the activity.
- Set the levels of support that the system will provide.
- Select the basic form of solution to the design problem.

A large number of architectures for Interactive Software have been described, e.g., Seeheim model, Model-View-Controller (MVC), Arch/Slinky, Presentation Abstraction Control (PAC), PAC-Amodeus, Model-View-Presenter (MVP). Most of these architectures are based on the traditional view is separated form the application model. The application part contains the functionality of the software and the view (user interface) part contains the representation of this functionality to the user(s) of the system. The motivation behind these architectures is to improve, among others, portability, reusability, multiple interfaces, customization, adaptability, usability, complexity handling, and separation of concerns of interactive software.

1.5 Software architecture versus designs

Software Architecture provides a view of the whole system. This distinguishes architecture from other design models, which focus on parts of a system. There are mainly two kinds of architecture – Monolithic and Reference. *Monolithic Architecture* is an architecture where all system functions (presentation, application and dialog) are tangled together in a single module. *Reference architecture* is a division of functionality plus data flows between the pieces and provides a common basis for communication about the domain (a common reference).

Within the envelope of the architecture, and in order to implement the architecture itself, we implement the design, which cares decomposition into components/modules/objects, specification of interfaces and design of protocols and formats. The outputs of the design process are typically expressed in UML diagrams, IDL specifications and XML DTDs. Software architecture is different from (Table 2) design model in several ways.

Table 2: Difference table of software architecture and design

Software Architecture	Software design
Architecture is just preliminary design where the primary structures of a software system are elaborated.	Software design is to take design decisions that are strategic in nature – those that define the primary system properties and provide a context for more detailed design decisions.
Architecture provides a view of the whole system.	Design focuses on parts of a system.
Architecture focuses on three aspects of software design: Partitioning – the functional partitioning of software modules, Interfaces – the software interfaces between modules and connection – the selection and characteristics of the technology used to implement the interface connections between software modules.	Design focuses to decomposition into components/modules/objects, specification of interfaces, and design of protocols and formats.
Architect is responsible for managing complexity.	Software design can formulate design rules that indicate good and bad combinations of choices. Such rules can be used to select an appropriate system design based on functional requirements.

1.6 Non-functional Qualities and Architecture

Now, we explore the relationship between the software architecture of a system and the non-functional qualities to be achieved by that system. We argue that there is an intimate connection between a system's architecture and the achievability of particular non-functional qualities within that system. The non-functional qualities of the system are orthogonal to the functionality of the

system. We view the functionality as the mapping of input to output generated by the system. This (admittedly narrow) view of functionality throws into the non-functional realm any discussion of performance, modifiability, portability, scalability, and so on.

Unit operations are a standard collection of structure-modifying techniques – separation, abstraction, compression, decomposition (part-whole and is-a), resource sharing and replication. Unit operations are applied on Software Architecture to achieve desired non-functional quality attributes. Unit operations are different form design patterns – they are more primitive. Design patterns are composed of unit operations. Unit operations are also more abstract than design patterns.

In order to construct complete software architecture we take the following steps:

- Determine the set of quality attributes and understand the ramifications of each quality of interest.
- Determine the functionality, which the architecture must compute to achieve each quality attribute.
- Make the architecture to compute each functions.
- Applying unit operations to the functions according to non-functional requirements.

Architecture serves both technical and organizational purposes. From the organizational perspective, the architecture helps to:

- *Understand the high-level design view*: A number of stakeholders need to understand the system at a fairly gross level. Modeling the system at a high level facilitates communication of the high-level system design. The reduction in detail makes it easier to grasp the assignment of significant system responsibilities to high-level structures.
- *Give idea about the system context*: The developers and future maintainers also need to understand the system at a gross level. In large systems, developers cannot efficiently understand the details of the entire system. Sometime, they need a detailed understanding of the more narrowly scoped portions of the system that they work on.
- *Allocate tasks*: Where architectures decompose the system into subsystems that are relatively independent, have clear responsibilities, and communicate with each other through a limited number of well-defined interfaces; the development work can be partitioned effectively. This allows parallel development work to proceed in relative independence between integration points.

At the technical level, architecture allows us to design better systems to:

- *Get system requirements and objectives*: Both functional and non-functional requirements can be prioritized as ``must have" vs. ``high want" vs. ``want", where ``must have" identifies properties that the system must have in order to

be acceptable. Architecture allows us to evaluate and make tradeoffs among requirements of differing priority. Though system qualities (also known as non-functional requirements) can be compromised later in the development process, many will not be met if not explicitly taken into account at the architectural level.

- *Enable flexible distribution/partitioning of the system*: A good architecture enables flexible distribution of the system by allowing the system and its constituent applications to be partitioned among processors in many different ways without having to redesign the distributable component parts. This requires careful attention to the distribution potential of components early in the architectural design process.
- *Reduce cost of maintenance and evolution*: Architecture can help minimize the costs of maintaining and evolving a given system over its entire lifetime by anticipating the main kinds of changes that will occur in the system, ensuring that the system's overall design will facilitate such changes, and localizing as far as possible the effects of such changes on design documents, code, and other system work products. This can be achieved by the minimization and control of subsystem interdependencies.
- *Increase reuse and integrate with legacy and third party software*: An architecture may be designed to enable and facilitate the (re)use of certain existing components, frameworks, class libraries, legacy or third-party applications, etc.

1.7 Design decisions made by developers that affect usability

Every software system has two aspects: physical and cognitive (Bevan, 1999). Others called them differently such as interaction and presentation. Physical aspect means providing right interface between user and system – user interface (UI). Cognitive aspect means matching the functionality, terminology, information and interface to the needs of the individual user. In 1980s and early 1990s the system design has traditionally been on building systems that meet specific functional requirements, without a sufficiently detailed understanding of the cognitive and physical capabilities and expectation of the individual users, or clear view of the context in which the system will be used. That is why most of the system has lack of Usability. On the other hand, architectures of the 1980s and 1990s assumed that usability was primarily a property of the presentation of the information. Therefore, simply separating the presentation from the dialog and application made it easy to modify that presentation after user testing. However, that assumption proved insufficient to achieve usable systems. A more popular belief in the 1990s was that usability concerns greatly affected system functionality (application) as well as the presentation (Len Bass, 2001). So achieving the correct functionality for a given system become paramount.

Len Bass and his team observed that if the presentation and the functionality of a system are designed extremely well, the usability of a system could be greatly compromised if the underlying architecture does not support user concerns. On

the other hand, if many modifications come to the fore after an initial design and implementation make the system unusable.

1.8 Objectives of this research

It is well know in software life cycle – the later a problem is detected; the more expensive it is to fix. Therefore, we will show in our thesis that the cause/effect relationship between internal and external attributes is a logical consequence of the Cartesian separation between the UI and application. The communication between application developers and interaction designers is the ideal place where cause/effect relationship occurs. The lack of communication between application developers and UI designers leads to low-level cause/effect relationships that affect the usability of the software. On the other hand, high-level cause/effect relationships between user requirements and the design decisions lead to a better usability of the software system.

This work aims to solve the problems of cause/effect relationship through an extension of the MVC model called MVCforUsability and in general through software architecture. Our goal is to define methods for usability engineering that can be integrated to an incremental and iterative software design process. A commitment to usability problems of a software system needs to put an emphasis on how and why the usability has to be improved. The answer to the why – is to increase the acceptance of the software system to the end-user and the answer to the how – is to integrate the user requirements in design process through the cause/effect relationship model between internal and external

attributes of software system. Here we have considered usability attributes - Learnability, memorability, speed of performance, error rate, satisfaction, and task completion as external attributes, and for each external attribute there are at least one invisible force which creates cause, we treat them as internal attributes.

2 Background and Related work

Our work has its foundation in interactive system architectures where several models have been proposed such as Seeheim, MVC, PAC, etc. Most of these architectures are based on the traditional view of interactive software can be separated form the core-functionality of the system. The motivation behind these architectures is to improve, among others, adaptability, portability, usability, complexity handling, and separation of concerns of interactive software as we discuss in this chapter. The principle of separating interactive software in application and user interface parts has its merits, but it can however lead to serious adaptability and usability problems in software that provides fast, frequent and intensive semantic feedback. Most of these models do not provide explicit solution for the cause/effect relationship between internal and external attributes of the software system to increase the usability of the software.

2.1 Seeheim Model

The Seeheim Model (Figure 1) has been established by X/Open Technology as a framework for a User Interface Management System (UIMS). It separated the whole software system into three layers. The layers are as follows:

Presentation Layer: Lexical aspect of the interaction. It is static, visible part of the interface built upon the X Window System and X toolkits, such as Xt Intrinsics and OSF/Motif.

Dialog Layer: Syntactic aspects of the interaction. It is responsible for the dynamics of the application. The dynamic portion that handles events (callbacks) and interfaces between the static screens and the application.

Application Layer: Semantic aspects of the interaction. It is the underlying application "functionality" that the GUI controls or communicates with. The application can be written in programming languages such as C, C++, Java or Ada or can even be an SQL-driven database.

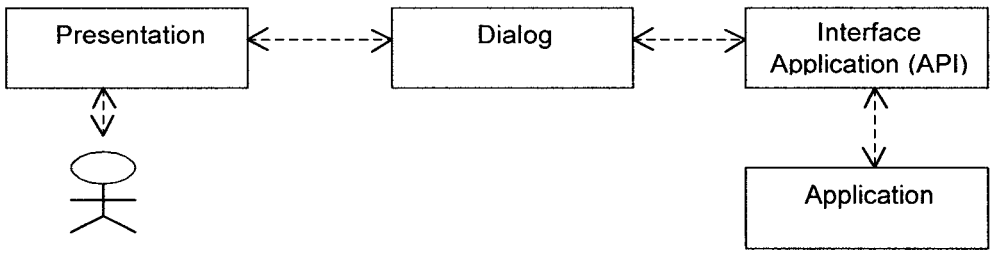

Figure 1. Seeheim conceptual model

Some weaknesses of Seeheim model are as follows:

- Many modifications affect all three functions – presentation, dialogue and application.
- Performance problems with sophisticated semantic feedback (e.g. input validation).
- Cumbersome to maintain separate notions for dialog, presentation and application layers.

2.2 MVC Model

One of the contributions of Xerox PARC to the art of programming is the multi-windowed highly interactive Smalltalk-80 interface. This type of interface has since been borrowed by the developers of the Apple Lisa and Macintosh and, in turn, by the Macintosh's many imitators. The central concept behind the Smalltalk-80 user interface is the *Model-View-Controller* (MVC) triad.

The Model-View-Controller (MVC) architecture (Figure 2) has three main components: the model, the view, and the controller. The model represents the underlying information of a specific user interface element. The view displays this information in a certain way, while the controller knows how the user interactions with the view will affect the information in the model.

In the MVC model the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object. The *view* manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application. The *controller* interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate. Finally, the *model* manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). The separation of these three components is an important notion that is particularly suited to Smalltalk-80 where the basic

behavior can be embodied in abstract objects: *View, Controller, Model* and *Object*. The MVC behavior is then inherited, added to, and modified as necessary to provide a flexible and powerful system.

MVC uses other design patterns, such as Factory Method to specify the default controller for a view, Decorator to add scrolling to a view and Observer to support multiple views.

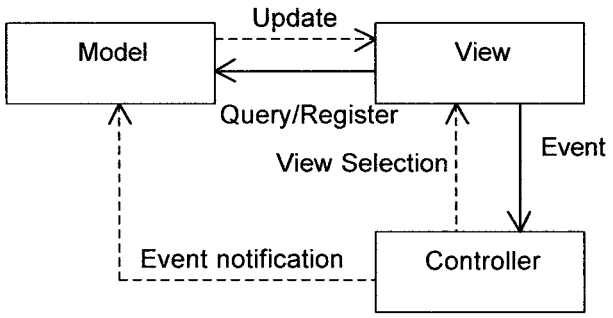

Figure 2. A pictorial representation of MVC architecture

Some of the major weaknesses of MVC Model are:

- Dependency mechanism may lead to a spaghetti of links– difficult to debug.
- Model is not well-developed– no notion of interface- application separation.
- the 'application' consists of one or more model objects; these are linked directly (tightly coupled) to the interface components (view and controller).

2.3 Comparison between Seeheim and MVC Model

Seeheim and MVC presented two fundamentally different approaches to dealing with modifiability. The following table (Table 3) presents major differences between MVC and PAC models.

Table 3: Comparison between MVC and Seeheim model

Seeheim	MVC
It presents three components: presentation, dialog and application to represent the interaction styles, some control knowledge of synchronization and the data.	It provides three low-level: model, view and controller to represent functional-core, presentation and control.
Seeheim presentation layer cannot be expressed in hierarchical order.	MVC view component can be expressed in hierarchical order.
Guard against changes is layering – placing distinct classes of functionality into distinct layers.	Guard against changes is part-whole decomposition – placing different pieces of system functionality, along with their input, output and dialogue, into distinct components.
Seeheim model offers a monolithic functional-core (Application) component.	MVC model usually distributes the functional-core (Model) into objects.
Seeheim allows either externally or internally controlled user interfaces.	MVC allows only externally controlled user interfaces, because each component is independent of each other. MVC uses a dependency manager to allow also limited internal control.
User Interface design is often based on components and structures provided for user interaction and task processing. Seeheim model does not support abstraction hierarchies.	MVC framework has rich possibilities for defining structural concepts by means of abstraction hierarchies.

2.4 PAC Model

PAC (Figure 3), for Presentation-Abstraction-Control, defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. PAC is a multi-agent model, it structures an interactive system as a collection of specialized computational units called agents. An agent has a state, possesses an expertise, and is capable of initiating and reacting to specific events. Every agent is responsible for a specific aspect of the application's functionality. It consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interface aspects of the agent from its functional core and its communication with other agents.

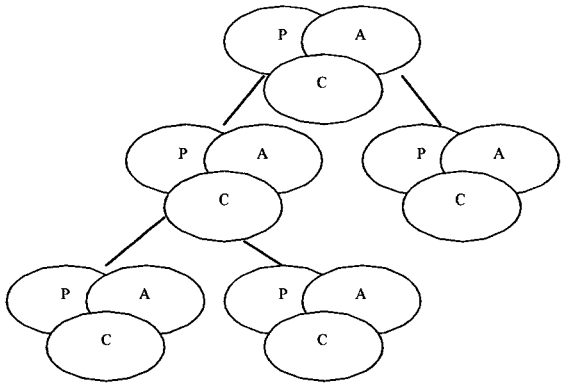

Figure 3. PAC architecture

PAC organizes an application as a tree-like hierarchy of PAC agents. There should be one top-level agent, several intermediate-level agents, and even more bottom-level agents. The whole hierarchy reflects transitive dependencies between agents. Each agent depends on all higher-level agents up the hierarchy to the top-level agent. The agent's presentation component provides the visible

behavior of the PAC agent. Its abstraction component maintains the data model that underlies the agent, and provides functionality that operates on this data. Its control component connects the presentation and abstraction components, and provides functionality that allows the agent to communicate with other PAC agents. The top-level PAC agent provides the functional core of the system. Bottom-level PAC agents represent self-contained semantic concepts on which users of the system can act, such as spreadsheets. Intermediate-level PAC agents represent either combinations of, or relationships between, lower-level agents.

Major Weakness of PAC Model:

A hierarchical PAC component makes a complicated composite PAC model.

2.5 Comparison between MVC and PAC model

The following table (Table 4) presents major differences between MVC and PAC models.

Table 4: Comparison between MVC and PAC models

MVC	PAC
In MVC the presentation (view) component corresponds to the low-level combination of view and control. Lack of de-coupling between components.	In PAC the presentation component is termed presentation. It provides better de-coupling between components.
MVC model is a part-whole decomposition.	The hierarchical order of PAC models supports composite PAC model.

MVC	PAC
In MVC model the dialog control component is embedded in the controller and controller manager components. There is no explicit component for ensuring consistency between model and view.	In PAC, the dialog control component is called control. It provides consistency for abstraction (application model) and presentation components.
MVC allows only externally controlled user interfaces, because each component is independent of each other. MVC uses a dependency manager to allow also limited internal control.	PAC allows external or internal control within a single interactive object because it distributes functional-core (Abstraction) in different objects.

2.6 Weaknesses of existing architectures

Most of the architectures are based on the separation between view and application model. The communication between view and application model makes the software system highly coupled as it becomes complex. These architectures also did not think how to integrate the usability concerns in design process.

2.6.1 Lack of usability

Simply separating the view from core functionality (Model) of the application and modify the view after user testing to increase the usability of the software proved insufficient to achieve usable systems. Recent research demonstrated that usability concerns are greatly affected by system functionality (application model) as well as the views (Bass, 2001).

2.6.2 Fallacious dichotomy between views and model

The principle of separation of interactive software into application and IU parts has its merits. It can however lead to serious usability problems (e.g., semantic feedback, adaptability, and portability). For example - semantic feedback (continuous feedback) makes application and UI to be highly coupled. Today, different people build the model and views. Example: multi-device user interfaces.

2.7 Promising Related Work under Investigations

Now we will investigate what researchers are thinking about usability, how they are integrating usability in interactive system architectures.

2.7.1 Len Bass Framework

Len Bass and his team presented (Len Bass and et al., 2001) an approach to improving the usability of software systems by means of software architectural decisions. They have identified specific connections between aspects of usability, such as the ability to "undo" and software architecture. They also formulated each aspect of usability as a scenario with a characteristic stimulus (event) and response (process by event-handler). Their framework, *Achieving Usability through Software Architecture*, has the following steps:

Step 1. Identified architecturally sensitive usability scenarios.

Step 2. Determined usability benefit hierarchy.

Step 3. Categorized usability scenarios into usability benefit hierarchy.

Step 4. Determined a Software engineering hierarchy.

Step 5. Architectural patters for each usability scenarios and categorizes into engineering hierarchy.

Step 6. Determined a matrix with Benefit Hierarchy in one axis and Engineering Hierarchy in the other. Each cell contains the general usability scenarios that correspond to the mechanism and benefit hierarchies.

The matrix provides many benefits. The software design team can decide which usability benefits are most valued in a particular project, uses the matrix to focus on the general scenarios that can provide those benefits to see which are applicable to that project and then reads off the architectural mechanisms necessary to implement those scenarios.

2.7.2 User Interface Patterns – Dorin Sandu

An interactive system is a system, which provides the user interface and the support for human activity. In interactive system users can interact with the UI to get their job done easily, so the demand of UI become paramount. Dorin Sandu found, after building user interface more details are required to have application interact with its user interface in a consistent way. Dorin Sandu introduced new patterns such as Event Handler, Complete Update, and Multiple Update to build

flexible user interface and along with a set of patterns (Subforms Patterns) that promotes user interface reuse.

According to Dorin, if developers use MVC (or any similar models) to implement user interface, and the visual components in user interface are already implemented on MVC model, then developers need to set dependencies, via observer pattern, between the user interface model and the visual components. In other words, each visual component needs to observe some aspect of the user interface model, and update itself when that model aspect changed. Then developers will find the resulting dependency graph is hard to understand and maintain.

To solve these problems, Dorin suggested some higher-level design decisions that developers can apply along with these patterns (MVC or similar patterns) to improve user interface design and implementation. Dorin incorporated these decisions into the patterns, mainly Event Handler, Complete Update, Multiple Update and a set of Subforms. We can describe them (Table 5) as follows:

Table 5: Some patterns used in Dorin's framework

Pattern	Problem	Solution
MVC	Architecture of Interactive system.	Divide into three components: model, view and controller
Event Handler	How should a view (observer visual component) handle an event notification message from its observable visual components?	Create and resister a handler method for each event from observable visual components.
Complete Update	How to implement behavior in the user interface to update the (observer) visual component from the model?	Assume all (observer) visual components are out-of-date and update everything.
Multiple Update	How to implement changes in the model of sub-form reflect to parent of sub-form, child of sub-form, siblings of sub-form?	Each sub-form should notify its parent when it changes the model. The parent should react to changes in the sub-form via Event Handler and update its children components via Complete Update.
Subfrom	How to design parts of user interfaces to operate on some model aspect?	Groups the components that operate on the same model aspect into subforms.

Event Handler, Complete Update, and Multiple Update 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 second phase updates the visual components to reflect the changes in the user interface

model. Since the update phase immediately follows the handle phase, the user interface always reflects the latest changes.

2.7.3 Commonly used patterns in interactive system design

Observer

The observer pattern defines a one to many dependencies between objects so that when one object changes state, all its dependents are notified and updated automatically. The Observer pattern manages the notification and updating between dependent objects called observers and subjects (observable objects). It describes how to notify observer objects when a subject changes its state, without forcing the subject to know the specific classes of the observers. MVC uses Observer to de-couple the model from the view, when the model changes, it notifies all its views to update. In this way, there can be many views operating on the same model at the same time, without the model explicitly knowing about all of them.

Command Action

It encapsulates requests for service from an object inside other objects, thereby letting us manipulate the requests in various ways.

We can use the Command pattern when:

- We want to implement a callback function capability. A callback function is a function that is made known (registered) to the system to be called at a later time when certain events occur.
- We wan to specify, queue, and execute requests at different times.

- We need to support undo and change log operations.

For example - toolkit objects provide a mechanism for invoking an operation, but the toolkit has no knowledge of the operation. In window buttons and menus: the toolkit provides the mechanism for activating an operation, but the application must supply the actual operation.

Abstract Factory

Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes (e.g. The Toolkit class). If we are 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.

2.8 Difference and originality of our objectives to existing ideas

In Len Bass framework, they have generated the list of usability scenarios by surveying the literature, by personal experience and by asking colleagues [Gram 1996, Newman 1995, and Nielsen 1993]. They did not establish the mechanism of getting usability scenarios. So, there is no clear indication how these scenarios are achieved. On the other hand, we have showed in our research how each usability scenario is achieved through the relationship between internal and

external attributes of the software system and we have tried to solve each scenario on the concept of separation as other framework did.

The main objective of Dorin's User Interface Patterns is to provide a mechanism to build, modify and maintain UI easily. Dorin showed how to handle events generated from the UI components, how to get changed object itself when UI model aspect has been changed and how to reuse the user interface. So, we can say these patterns are UI centric. Dorin did not think about the usability of the user interface at all.

3 Proposed Methodological Framework

The framework (MVCforUsability) we are proposing is based on Len Bass work [2001]. It has four different steps for solving cause/effect relationship problems between internal and external attributes of the software system and to integrate usability in design process.

1. Identify the internal software attributes that affect software usability, such as modularity, functionality, etc.
2. Define the relationship between these internal attributes and external usability factors in terms of typical scenarios.
3. Describe design patterns or improve existing ones for each typical scenario.
4. Discuss the usage of such patterns within the MVC framework.

Now we will explain different steps in detail in the following chapters.

3.1 Identify internal attributes that affect software usability

As we know internal attributes of an entity are those that can be measured in terms of the entity itself, generally they are not either visible to the users or users' concern. It is the designers and developers concern what kind of internal attributes they would account to integrate user requirements in their proposed software system. It is very important to get the correct form of internal attributes to implement user requirements.

The following are some examples of high-level internal attributes to implement user primary requirements and to achieve user goals.

1. *UI component familiarity mechanism (Leverage Human Knowledge):* Users use what they already know when they are approaching a new situation. If the 'look and feel' of new software components/objects are similar to what user has already seen or used then user can create metaphor in his/her mind about the functionality of the specific component and take initiative to use it.

2. *Natural Mapping:* Creates a clear relationship between what the user wants to do and the mechanism for doing it. Example - to perform my task, I need to select this option, enter that information, and then press this button.

3. *Data and commands Aggregation:* When users want to perform action on more then one object then system aggregate data into one object and perform the action on new object for short cut. In the same way, when it takes several commands to achieve one goal then system aggregate commands into new object for short cut.

4. *Distinct views with same functionality mechanism:* User would be confused by functional deviations among different views of the same data. So commands should be available for all views of the same data. Example: UNIX windows.

5. *Undo/cancel mechanism:* Undo is an action, which is used when user no longer wants the effect of the operations. For example, a user accidentally deletes a file and wishes to restore it. Cancel is an action, which is used when user invokes an operation, but no longer wants the operation to be performed. For example, a user selects an unintended menu item due to slip of mouse, but no longer wants the operation to be performed.

6. *Recognize user error mechanism (spell checking):* A user could make an error that he or she does not notice. Sometime human error is predictable through perceptual and cognitive analysis.

7. *Customizability mechanism:* Customization is the ability to change the user interface (UI) to adapt with the user habits and environment. The degree to which users can customize software system to suit their own environment is a critical factor.

8. *Robustness mechanism (fault tolerant):* Robustness has many meanings in many contexts. Robustness is the ability of a system with a fixed structure to perform multiple functional tasks as needed in a changing environment. Robust software will be able to better accommodate evolutionary change by anticipating and isolating concerns in the original design and implementation. For example – user wants to install a new device; the device may conflict with other devices already present in the system.

9. *Display system status mechanism (feedback):* System status is the situation what system is doing when user invokes command. User may not be presented with system state data necessary to execute the command, but system should let the user know how much work has been done to accomplish the goal.

10. *Support multiple-visualization mechanism:* Multiple visualization means to see objects in different shape and structure. Sometimes user needs to see data in different shape and structure to understand and analyze them.

3.2 External software attribute

External attributes of an entity are those that can be measured only with respect to how the entity relates to its environment, i.e., by observing its behavior in its environment (e.g. execution time of a program on a particular machine). External attributes are often measured in terms of internal attributes. There are many external attributes like usability, reliability, efficiency, testability, reusability, portability, interoperability, and understandability just to mention some. Here we will consider only usability attributes as external attributes because our goal is to integrate usability in design process.

Usability means how easily the end users reach their targets and how satisfied they are with using the product (ISO/FDIS 13407, 1999). A set of attributes of

software which bears on the effort needed for use and on the individual assessment of such use by a stated or implied set of users (ISO/IEC 9126: 1991). Usability is measuring how well a system supports user's activity. The following are some common usability attributes we called them external attributes of the system:

1. Learn system features (Learnability).
2. Use system efficiently (Efficiency).
3. Minimize the impact of errors (Error Minimization).
4. Adapt the system (Adaptability).
5. Feel comfortable (Comfortability).

Many other attributes are also suggested in the literature (Donyaee, 2000).

3.3 Cause/Effect Relationship Scenarios

Formally speaking, the cause (Internal attributes) and effect (External attributes) relationship can be expressed as follow:

Relation R (Unit operation, quality attribute) = R (Internal attributes, External attributes) = R (X1, X2) = Typical scenario

We use the concept of scenario to describe a cause/effect relationship. A scenario is story that describes a typical situation where a specific cause/effect relationship occurs. We use the following examples to explain typical scenario.

Example 1:

If user wants to bold a paragraph, first he/she needs to select the paragraph (Data aggregation) which to be bolded and passes this aggregated data through

the BoldFunction(aggregatedData) function as parameter to get bolded paragraph. In this way, user can bold as much text as he/she wants at one shot, so that user can expedite workflow and minimize the error. Here we can treat *'data aggregation'* as internal attribute, which creates 'cause' and *'work efficiency'* as external attribute, which is 'effect' of the cause. It creates a scenario, which needs to be implemented through the internal attribute to increase efficiency and decrease user errors.

Example 2:

A user may make an error that he/she does not notice. However, human error can frequently be circumscribed by predictable perceptual, cognitive analysis and the nature of the task at hand. Users often type 'hte' and 'fo' instead of 'the' and 'of' in word processors. The frequency of the word 'the' and 'of' in English and the fact that 'hte' and 'fo' is not a English word, combined with the frequency of typing errors that involve switching letters typed by alternate hands, make automatically correcting to 'the' and 'of' almost always appropriate. Computer-aided correction mechanism (module) can be used to detect user error and can provide help to correct them. Computer-aided correction can be either enforced directly for automatic text replacement or suggested through system prompts (feedback). This mechanism decreases the user error and increases the individual satisfaction and performance. Here we can treat *'computer-aided correction mechanism or feedback'* as internal attribute, which creates 'cause' and *'satisfaction and performance'* as external attribute, which is 'effect' of the

cause. It creates a scenario, which needs to be implemented to increase user satisfaction and performance, and decrease user errors.

Example 3:

The user always wants to know whether or not the operation is still being performed as well as how much longer the user will need to wait. When user invokes an operation, the system provides feedback that the application is still working and gives an indication to the progress to keep the user informed. These mechanisms indicate the system performance and give user satisfaction. Here we can treat *'Feedback and progress bar mechanism'* as internal attribute, which is 'cause' and *'performance and satisfaction'* as external attribute, which is 'effect' of the cause. It creates a scenario, which needs to be implemented to increase user satisfaction.

The scenario gives more details on how the internal attributes affect the usability factors (external attributes). External attributes are prevailing on the user environment; stakeholders need to express them in term of scenarios, which is called cause/effect relationship between internal and external attributes. On the other hand, to get any external attribute (usability attribute), designers need to design for related scenario and developers need to implement internal attributes related with the design. The following table (Table 6) shows some potential relationships between internal and external attributes.

Table 6: Potential relationship between internal and external attributes

External / Internal	Performance	Robustness	Customization	Reusability	Adaptability
Controllability		0	1		1
Immediate Feedback	1				0
Fault tolerance mechanism	0	1			0
Recovery mechanism	0	0			0
Modularity	0			1	

1 Strong relationship, very common

0 Potential relationship to be validated

As part of our thesis, we identified the following 10 scenarios:

- Create Metaphor = R (UI component familiarity mechanism, Learnability).
- Comparability = R (Natural Mapping, Learnability).
- Decrease user error and increase performance = R (Data and commands aggregation, efficiency).
- Operating consistently across views = R (Distinct views with same functionality mechanism, efficiency).
- Error Recovery = R (Undo/cancel mechanism, error minimization).
- Checking for correctness = R (Recognize user error mechanism, error minimization).

- Providing user choices = R (Customization mechanism, adaptability).
- Providing device independent = R (Robustness mechanism, adaptability).
- Provide user confidence = R (Display system status (feedback), comfortability).
- Working Data Visualization = R (Support multiple visualization mechanism, comfortability).

3.3.1 Create Metaphor = R (UI component familiarity mechanism, Learnability)

Here we consider 'UI components/objects familiarity' as internal attribute and 'Learnability' as external attribute. This relation creates a scenario that needs to be implemented so that users can have ability to figure out how to use something just by looking at it and can perceive the properties of the components of UI. *Example:* If we see a handle, we are tempted to pull it or if we see a switch, we are tempted to set it in the opposite direction. Create metaphor of an object is the action it intuitively tells us to perform on it.

3.3.2 Comparability = R (Natural Mapping, Learnability)

Here we consider 'Natural mapping' as internal attribute and Learnability' as external attribute. This relation creates a scenario that needs to be implemented so that users can have ability to figure out how to achieve a goal. *Example:* Mapping is the technical term for associating one thing with another. In the sense of Usability, this is mostly seen as the result of an action. One of the most

common mappings is the one of the steering wheel in a car. When we turn the steering wheel to the left, our car turns left. Steering to the right causes us to move right. This mapping, however, does not hold if we drive a car with a trailer in reverse. We have to steer the opposite way, to get the trailer moving the way we want it to.

The first mapping mentioned is a natural mapping and the other is arbitrary. A natural mapping takes advantage of some physical analogy or a cultural standard. When designing objects, we should use natural mapping as much as possible. Natural mapping aid the user tremendously in learning the application. Furthermore, through natural mapping the object or tool seems to work and behave intuitively.

3.3.3 Decrease user error and Increase performance = R (Data and commands aggregation, efficiency)

Here we consider 'Data and commands aggregation' as internal attribute and 'use system efficiently' as external attribute. This relation creates a scenario, which needs to be implemented so that users can have ability to figure out how to increase performance and decrease his/her errors.

Data Aggregation

Data aggregation is any process in which information is gathered to pass as a parameter to the functions to do some manipulation with aggregated data. *Example:* If user wants some portion of text in a Text Editor to bold, user needs

to select the data to be bold and to pass the selected data as parameter to the BoldFunction(selectedData) to get the selected (aggregated) data bolded.

Command aggregation

Aggregation allows a command to be played and then absorbed by the command at the top of the command stack. *Example:* Replace all - a replace all command is first created, then a series of replace commands are executed against the document and each of these commands is aggregated into the replace all command.

3.3.4 Operating consistently across views = R (Distinct views with same functionality mechanism, efficiency)

Here we consider 'Distinct views with same functionality mechanism' as internal attribute and 'use system efficiently' as external attribute. This relation creates a scenario which needs to be implemented so that users will have same functionality for different view of the same data. *Example:* In UNIX user can open up as many windows as he/she wishes and from each windows user can apply same commands on same data. There are no functional deviations between different views of the same data.

3.3.5 Error Recovery = R (Undo/cancel mechanism, error minimization)

Here we consider 'Undo/cancel mechanism' as internal attribute and 'error minimization' as external attribute. This relation creates a scenario which needs

to be implemented so that users can try to figure out how to accomplish a task without any fear and hesitation; and can have flexibility to change his/her mind according to the situation. *Example:* Dix, Finlay, Abowd, & Beale (Dix, et al., 1993) make a distinction between backward and forward error recovery. This distinction is now commonly used in interactive system development.

Backward error recovery

Backward error recovery is an attempt to restore the System State after an error has been occurred. Backward recovery can be considered as the only real "recovery" function, since the unexpected effects of error are totally removed. Backward error recovery function is to go back in time. According to Yang (Yang, 1992) – in (Lenman & Robert, 1994a) – there are three kinds of backward error recovery commands: *undo*, *cancel*, and *stop*.

- The *undo* function is the most famous one, and most editors implement an undo function. Designers need to think with the presentation, the granularity, the scope, and the range of the undo function.
- The *stop* function is used to terminate the process under execution. For example, a user can make the choice to stop a long printing command when he realizes that many users are waiting for the printer.
- The *cancel* function is used to abandon commands under context. For example, a user can cancel the typing of an e-mail message if the addressee has just entered the user's office. The *cancel* function needs to deal with its scope.

Forward error recovery

In forward error recovery, the user has to execute unexpected tasks to recover the fault. For example, if you break a dish plate, you have to use glue to recover your error. Of course, the final dish plate is not as nice as the unbroken one.

3.3.6 Checking for correctness = R (Recognize user error mechanism, error minimization)

Here we consider 'Recognize user error mechanism' as internal attribute and 'error minimization' as external attribute. This relation creates a scenario which needs to be implemented so that if users make any mistake system will let the user know what he/she has done wrong and what they need to do to achieve the goals by providing some possible ways. *Example:* There are two kinds of correctness – well-formedness and validity. Well-formedness checks spelling and meaning of the sentence (spell checking). Validity checks the inputs against Pre-defined Data Type (PDT)

3.3.7 Providing user choices = R (Customization mechanism, adaptability)

Here we consider 'Customizability mechanism' as internal attribute and 'adaptability' as external attribute. This relation creates a scenario, which needs to be implemented so that users can have ability to change the look and feel of the system, especially UI. *Example:* Customization is the ability to maximize the use of what software already has to meet the user needs. In other words,

customization is the way to enhance user profile; using a 'user profile' the system constructs a relevant, dynamic, seasonally adjusted portal for user. There are four kinds of customization.

Adaptive Customization: No change in either product or representation, user can filter out most of the possibilities using pop-up menus, search functions and preference settings.

Cosmetic Customisation: A different presentation of a standard product.

Transparent Customisation: Change in product, but not in representation.

Collaborative Customisation: Change in both product and representation.

3.3.8 Providing device independent = R (Robustness mechanism, adaptability)

Here we consider 'Robustness mechanism' as internal attribute and 'adaptability' as external attribute. This relation creates a scenario which needs to be implemented so that software system will have the adaptability in different platforms and environments. *Example:* Robustness is the degree to which a software component functions correctly in the presence of exceptional inputs or stressful environmental conditions. In the other words we can say - robustness is the degree to which a system or component can function correctly in the presence of invalid or conflicting inputs.

Programs fail mainly for two reasons: logic errors in the code, and exception failures. Exception failures can account for up to 2/3 of system crashes, hence are worthy of serious attention.

3.3.9 Provide user confidence = R (Display system status (feedback), comfortability)

Here we consider 'Display system status (feedback)' as internal attribute and 'comfortability' as external attribute. This relation creates a scenario that needs to be implemented so that users will have ability to know what system is doing and what they can do next. *Example:* The system prompts to the user indicate that a task is being done or the task is being doing on progress.

3.3.10 Working Data Visualization = R (Support multiple visualization mechanism, comfortability)

Here we consider 'Support multiple visualization mechanism' as internal attribute and 'comfortability' as external attribute. This relation creates a scenario which needs to be implemented so that data or content can be visualized in a meaningful ways. *Example:* Sometimes user wishes to see data from different point of views. In Microsoft PowerPoint user can see or show his/her working data in different ways to make his/her presentation more understandable to the audiences.

4 MVCforUsability – From Scenarios to Patterns

A pattern is a solution to a recurrent problem with set of forces (set of goals) in a context. Patterns were first used to describe shapes of structural elements in buildings and towns. For example, the "half-hidden garden" pattern considers how to frame a view of a garden so that onlookers have a sense of well-being and positive anticipation [Alexander 79].

Different "standards" have emerged to describe Software Patterns. The simplest description requires the following items:

- *Context* refers to a recurring set of situations in which the pattern applies.
- *Force* refers to a set of goals to be achieved by pattern.
- *Problem* refers to a set constraint and limitation that need to be overcome by pattern solutions.
- *Solution* refers to a canonical design form or design rule that someone can apply to *resolve* these forces.

Example: *Name*: Window Place; *context*: Design of a residential room; *forces*: People want to sit and also be in daylight; *problems*: If all seating is away form window people will be dissatisfied and *solution*: Build seating near window.

4.1 Creates metaphor

Name of Pattern: Create Metaphor

Context: The user always wants to understand the system with minimal affords and experiences. That means UI components or objects need to have two things: Visibility - to give the user the ability to figure out how to use something just by looking at it and *Affordance* – to make an idea about objects properties and how the objects are to be used.

Forces:

- Users need to predict about the meaning and functionality of the objects.
- Users need to see many objects but want to see them in a clear organized way.
- Each object creates non-overleaping idea, which maps with object actual functionality.
- Users want to minimize the time takes to scan/read/view objects on the screen.
- Users want objects are often to be related and can be grouped conceptually.
- Users want the presentation of the objects needs to be compact, but still clear, pleasant and readable.

Problems: If user interface does not fit with the histories in user mind user will not be able to create Mental Model about the functionality of the system.

Solution: Metaphor (Figure 4) is the structure-mapping from a source onto a target. Such mapping can exploit existing common schematic structure or new structure from the source onto the target. The work on the 'Conceptual integration' has shown that in addition to such mappings there are dynamic integration processes which build up new 'mental spaces'. Such spaces develop

emergent structure, which is elaborated in the on-line construction of meaning and serves as an important locus of cognitive activity.

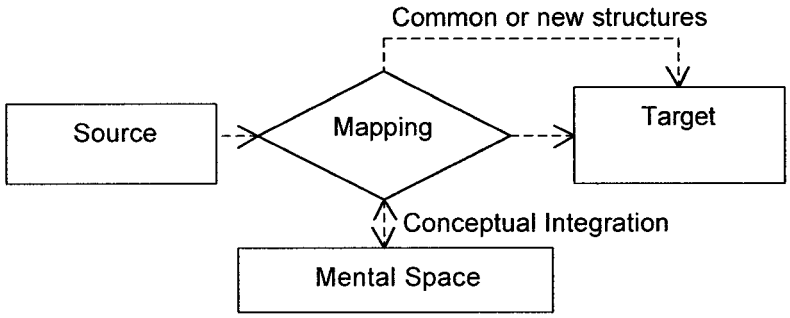

Figure 4: Conceptual (Logical) Pattern for Metaphor

The cross-space mapping between the inputs (from the source and the target) is metaphoric. But the blend (Conceptual integration) has causal and event shape structure that do not come from the source, indeed are contrary to the source and in some cases impossible for the source, and the central inference of the metaphor cannot be projected from the source.

Concrete Example: Open a file

Source: User intention (from user mind) is to get a file from a folder

Target: Icon of 'open folder' (a button or icon with a picture of 'open folder')

Mapping: Compare both structures conceptually – Map (Source structure, Target structure, Mental space structure) = Mental space structure

Mental Space: Task analysis is done how to achieve the goal. If user likes to open a new file, first user needs to find out the location of the file, and then user opens the desired file.

Mapping result: The Idea how to open a file.

4.1.1 Usage of such pattern within the MVC framework

To create metaphor in user mind about different objects' functionality of UI is depend on above conceptual Pattern. It would be best idea to separate UI components or objects from its functionality, and then we will be able to arrange or manipulate different object to create metaphor in user mind without affecting the functionality of the object. That means UI components have to treat as View, the functionality as Model and communication as Controller.

4.2 Comparability

Name of Pattern – Comparability (Keep user thinking logically)

Context: User wants a clear relationship between what user wants to do and the mechanism for doing it, which is called the Natural Mapping. On the other hand, user wants solutions for any circumstances where user needs to think what to do next or what would be the next steps.

Forces:

- The user wants to achieve the goal but may not be familiar or interested in the steps that need to be followed.
- The subtask should be ordered but are not always independent of each other i.e. a certain task may need to be finished before the next task can be done.
- To reach the goal, sometimes several steps need to be taken but the exact steps required may vary because of decisions made in previous steps.

Problems: If the tasks are not conceptually subdivided in step by step according to user conception, users will not be able to achieve their goals and will be frustrated.

Solution: Pattern matching (Figure 5) always involves an attempt to link two patterns where one is a theoretical pattern and the other is an operational one. The top part of the figure (Figure 5) is User General Ideas to perform a task. The Task Model creates general Conceptual Pattern how to achieve user Goal. The bottom part of the figure indicates the realm of Operational Pattern.

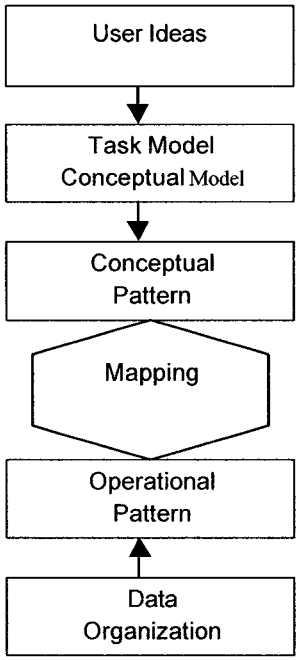

Figure 5: Comparability Pattern – creates natural mapping between users and objects

4.2.1 Usage of such patterns within the MVC framework

Comparability pattern links user conceptual ideas and the system operational mechanism through natural mapping. User conceptual ideas fully related with the UI components, on the other hand operational mechanism fully related with core-functionality of the system. So, we can capture conceptual ideas into UI objects (View) and operational mechanism into core-functionality (Model) of the system.

4.3 Decrease user error and increase performance

Name of Pattern: Decrease user error and increase performance

Context: Users may want to perform one or more actions on more than one object (data aggregation – user selects a paragraph of text to bold it up) at a time. On the other hand, user may want to complete multi-step procedure consisting of several commands at one shot (command aggregation – a batch of commands).

Forces:

- System should allow users to select and act upon arbitrary combinations of data and commands.
- The specific aggregations of actions or data that user wishes to perform cannot be predicted.
- System should provide a batch or macro capability to allow users to aggregate data and commands.

Problems: If the user cannot select arbitrary combination of data and commands it will be tedious to do one by one.

Solution:

Data aggregation pattern (Figure 6) has the following components:

Command Receiver - This component manages the commands that the user generates. A command has an action and one or more subjects that either provides input or accept output form the command. Command is passed to the Grouping Manager for creation a group, adding data to a group, removing data from a group.

Grouping Manager - This component manages the definition of groups and addition and deletion of data items from a group. Grouping Manager controls the iteration of commands through the Command Processor. It accesses the User Data to present group.

User Data - This component provides access to the application data that is visible to the user. These data are available both to those components that control data presentation and to those components that manipulate the data.

There are two options for applying a command to a group; these are iteration and embedded ways. In iteration way particular command operates on single argument and grouping manager repeatedly invokes the correct command processor on each item in the group. In embedded way Command Processor understands groups and can directly operate on the grouped data.

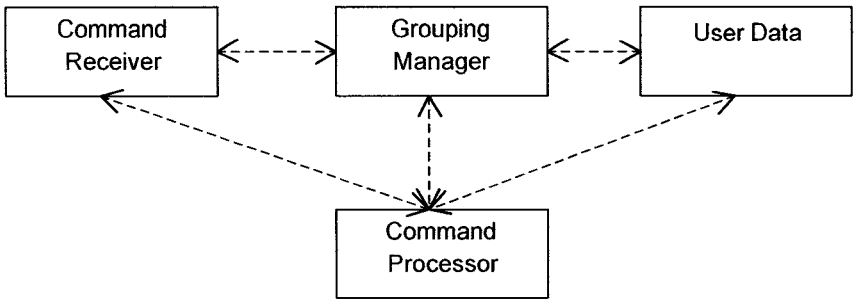

Figure 6: Data aggregation pattern

Command Aggregation pattern (Figure 7) has the following components:

Command Receiver - It receives user requests (commands) that are desired synchronously. It must communicate the commands both to the command processor for execution and to the authoring editor for inclusion in the aggregate. The authoring editor and the command processor may communicate if additional information about parameters or data must be saved in the aggregated command.

Authoring Editor - The authoring editor monitors commands that are received by the Command receiver and saves them as an aggregated set that can be subsequently invoked. These aggregated commands are usually edited prior to final saving for execution.

Execution - User invokes the aggregated command. The command receiver must communicate with the authoring editor to retrieve the aggregated command. This necessity for communication is the source of the data flow from the authoring editor to the command manager. The command receiver then sends the commands one at a time to the command processor. The command manager

must be informed of the source of any required synchronous user input and the destination of any generated error messages.

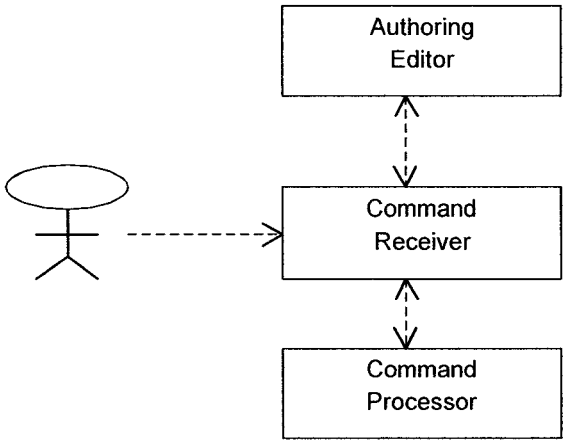

Figure 7: Command aggregation

4.3.1 Usage of such patterns within the MVC framework

Data aggregation: In data aggregation user aggregates the data from his/her editor and pass it as a parameter to the command invoked. So we can think 'user selected data' as View and the command implementation as the Model.

Command aggregation: Commands are aggregated in 'Authoring Editor' according to the user selection. So we can think 'Authoring Editor' of commands as View but each command implementation as Model.

4.4 Operating consistently across views

Name of Pattern: Operating consistently across views

Context: User wants operation consistency between views or presentations. If there are any functional deviations between different views for the same data

then user will be confused. So, system should make commands available based on the type and content of user's data, rather than the current view of that data.

Forces:

- Users like to have same functionality between different views or presentations for the same data.
- Users like to have same resources to execute their operations for different views.
- System should make commands available based on the type and content of user's data, rather than the current presentation (view) of that data, as long as those operations make sense in the current presentation.

Problems: If commands that have been available in one view may become unavailable in another or may require different access methods for the same data then user would be frustrated.

Solution: The 'Data Model' (Figure 8) is being viewed should be separated from the 'View descriptor' so that 'View' is independent from 'Data Model'.

'View' maps the data through 'View Descriptor'.

'Command controller' is separated from 'Data Model' so that user commands operate on 'Data Model' without knowing 'View'.

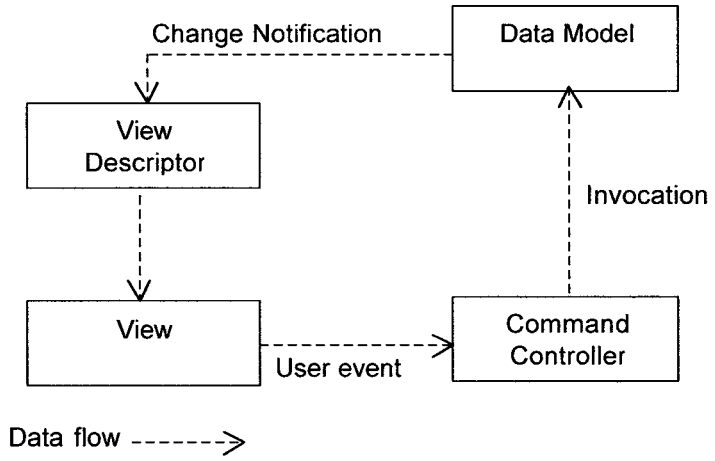

Figure 8: Make consistency Pattern

4.4.1 Usage of such patterns within the MVC framework

Here Data, View, Command and view descriptions are separated form each other. So, we can implement and change any component without affecting others like MVC model.

4.5 Error Recovery

In window-based GUI applications, it is standard to have a *Cancel* button that closes any dialog box and discards any changes the user may have made within that dialog box. It is a great way to support exploratory learning compare with older systems where we were trapped if we ever activated the wrong command. In editing systems it is usual to have an *Undo* command that makes the document revert to the state before the user's most recent changes. Sometimes, multi-level undo and redo is supported: this can be very useful but confusing. The following table (Table 7) summarizes the common differences between *Undo* and *Cancel*.

Table 7: Logical differences between Undo and Cancel actions

Undo	Cancel	Comment
Undo is being done on a performed operation.	Cancel is being done on an ongoing operation.	The time to use
The system allows the user to return to the state before the operation was performed.	The system stops invoking operation to be performed.	Consequences
The granularity of the undo (e.g. are keystrokes undone or commands) is dependent on the component rather than on some system-wide decision.	The granularity of the Cancel is a system-wide decision.	Action boundary

Undo

Name of Pattern: Undo

Context: To err is human - user can make a mistake. There should be a way to undo the mistaken operation, so that user won't loose anything for his/her accidental mistaken. So, software system needs to have a mechanism to return to the state before the operation is performed.

Forces:

- User wants to have choice to perform an operation and change his or her mind about wanting the effect of that operation.

- A user may accidentally delete some lines in a document and wish to restore it without lose of generality.
- Users are curious, they want to try something without knowing anything about the functionality of the objects if there is a way to get back.

Problems: User won't try if there is no way to get back. If user invokes Undo operation user might loose previous operation execution data, so user might not get prior state and will be frustrated.

Solution:

Undo Component (Figure 9):

- Each component sends relevant data to Undo Manager.
- Data are viewed as a transaction.
- Each transaction is stored as atomic unit.

Undo Manager:

- Stores each transaction in a stack as atomic unit.
- Using a global transaction manager rather than relying on each component to perform its own undo has the advantage that the number of steps that can be undone is arbitrary (multi-level undo).
- The granularity of the undo is dependent on the component rather than on system wide. Suppose the current granularity of the undo is at the command level and there is a decision made to change it to the keystroke level. Then all that is necessary is to enter the keystrokes into the Undo Manager and the commands do not need to be modified.

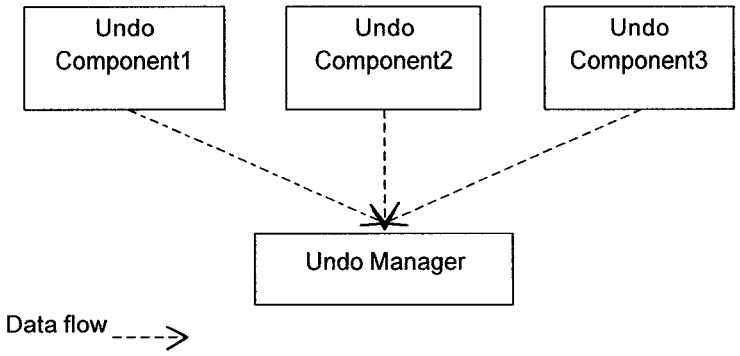

Figure 9: Undo Pattern

Cancel

Name of Pattern: Cancel

Context: When user is working in a computer and get a call to be rushed to an unexpected place, then user no longer wants the invoked operation to be performed. The user now wants to stop the operation rather than wait for it to complete. User could have clicked or selected one for another, system should allow user to cancel operations.

Forces:

- User invokes an operation, but no longer wants the operation to be performed.
- The user wants to stop the operation and go back to the prior state without loosing anything.
- User want to have prevention for accidental 'slip of click' or 'slip of select'.

Problems: If the user wishes to stop current operation to get back to prior state but system lost prior data then user will be frustrated.

Solution: Active Component (Figure 10): These components perform the activities that may be cancelled. They provide the information about the used

resources by them to the Controller. They keep sufficient information about the prior system state. They provide resources to Cancellation controller to cancel the command.

Cancel Listener: This component listens to the user request for canceling the active components passes it to the Cancellation Controller. It informs the user that it has received the cancellation request.

Cancellation Controller: This component terminates the active thread, and returns the persistent resources to their state prior invoking the active components, release non-preemptable resources, provide feedback to the user about progress and the result of the cancellation, and inform collaborating components of the termination of the active thread.

Collaborators: This component takes the information about terminated components.

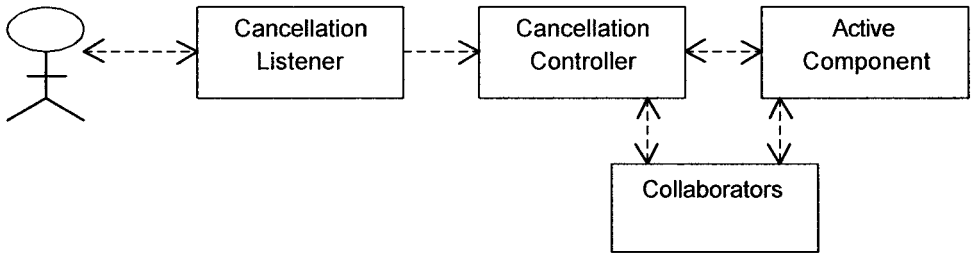

Figure 10: Cancel Pattern

4.5.1 Usage of such patterns within the MVC framework

Undo: 'Undo Manager' is separated form undoable functions, so this component is modifiable without affecting other components.

Cancel: Components are separated from each other, so that they can be modified independently.

4.6 Checking for correctness

Name of the Pattern: Checking for correctness

Context: Human error can be detected by nature of the task at hand. For example, users often type 'hte' instead of 'the' in word editor. These error corrections can be done in two ways – automatic replacement and suggestion through system prompts.

Forces:

- User wants his/her errors will be detected by the system and take action through either automatic replacement or suggested by system prompts.
- System prompts should be clear with possible answers so that user will not be confused.

Problems: If user error is not detected by the system then user need to take initiative to figure out what is the cause of the error and if fails he/she will be frustrated.

Solution:

User Input (Figure 11): We can think it as Application Data. It should be separated from Checker so that application data can be accessible by the checker.

Checker: It has different models in its mind – Task Model, User Model and System Model – so that it can determine when a potential error occurs and what will be the possible corrections.

Presentation: It only shows the corrected output. Here user data will be replaced by corrected output.

Task Model: This model determines what user is attempting to do to accomplish his/her task so Checker can take initiatives.

User Model: This model determines knowledge of the user so Checker can take initiatives.

System Model: This model determines expected behavior of the Application (System) so that system can provide dialogues to the user.

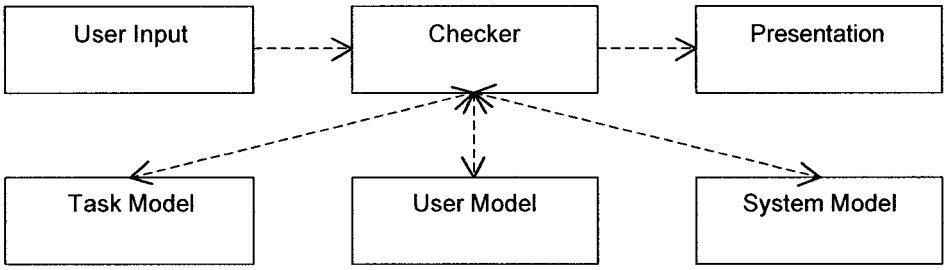

Figure 11: Checking Pattern

4.6.1 Usage of such patterns within the MVC framework

User inputs is separated form the presentations and checker, so presentations and checker can be modified without affecting each other.

4.7 Provide user choices

Name of Pattern: Provide user choices (Modifying Interface)

Context: Interactive system is the lifeblood to the users. So, uses should have the ability to modify the UI according the users choices because different users have different choices.

Forces:

- Different individuals have different style and choices. Each individual likes to work in his/her own ways.
- Some users have some disability problems so they like to adapt their system in own ways.
- Changing the look and feel; and functionality should be easy to understand and modify.

Problems: If users cannot modify UI according to their wishes they will be frustrated.

Solution: To support the modifiability we need to follow the following steps:

- Enumerate a list of likely change scenarios.
- Decide the functionality of the scenarios.
- Encapsulate each functionality.
- Indirection of both data and control.

Encapsulation (Figure 12): Encapsulate all user interface functionality away from the core of the system allows designers to modify the user interface more easily.

Indirection (function): The use of an intermediary such as the dialogue manager in the Seeheim Model or the Controller in the PAC model is indirection of function.

Indirection (data): Indirection of data refers to the separation of application data from the view of that data. It occurs in virtually.

Separate all functionality which will be modified by the user from unmodifiable functionality of the system. Then separate the data (Figure 12), controller and functionality of the modifiable functions to facilitate user-modifying talks.

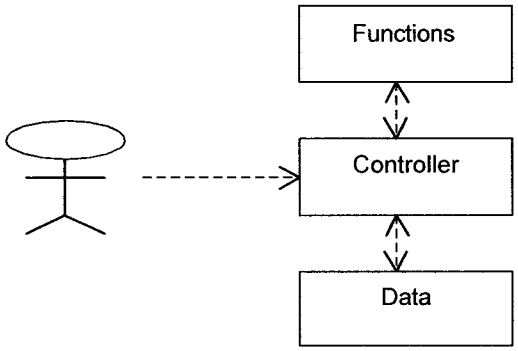

Figure 12: Modifying interface pattern

4.7.1 Usage of such patterns within the MVC framework

Function controller is fully separated form the implementation of the function and data. User can modify the controller of the function without affection implementation of the function and data.

4.8 Provide device independent

Name of Pattern: Provide device independent

Context: Any application should be device and platform independent, so that users don't need to think about the required environments. The more the system is robust, the more the system is conflict free with other existing software. Application should be designed to reduce the severity and frequency of device conflicts.

Forces:

- User wants his/her application supports all devices including to be installed in his/her machine. That means application will not conflict with any other software components exist in his/her machine.
- When there are any device conflicts occur, the system should provide the information necessary to either solve the problem or seek assistance.

Problems: If there are any device conflicts with new software component installation then the user will be frustrated.

Solution:

Application Layer (Figure 13): This is our proposed application

Virtual Layer: It acts like a mediator. It is an API. Application accesses Physical Devices through Virtual Layer, it defines and abstracts how the devices should be controlled. It accesses the Physical Devices through the physical device drivers that do the actual control. The function of the Virtual Layer is to translate the Virtual Layer into the various Physical Devices.

Physical Layer: A class of device drivers. For each class of devices there is an abstract Virtual Layer by which Application get contact with Physical devices.

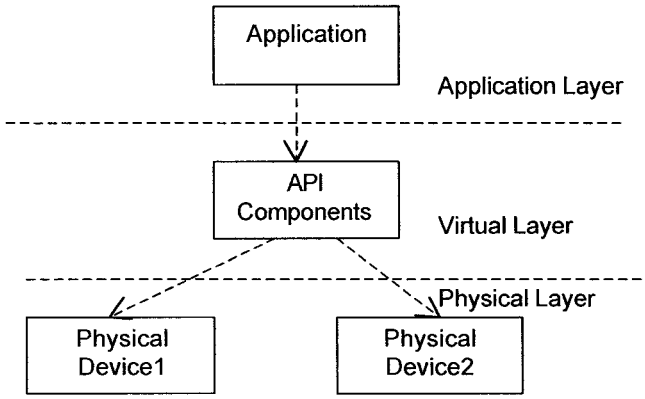

Figure 13: A logical representation of software components in the machine (Providing device independent pattern)

4.8.1 Usage of such patterns within the MVC framework

API is treated as a separate component, which is shared by both software system and device drivers and provides necessary resources to software system. API and software systems are fully independent components, so they are modifiable independently without affecting each other.

4.9 Provide user confidence

Name of Pattern: Provide user confidence (feedback)

Context: When a task is being completed the system gets different mode or state in different situations. User needs to be informed about what is going on to get user confidence.

Forces:

- The user cannot always control the performance of the operation, because it may rely on external system, which may fail, block or have low performance.

But user can express his/her own opinion with respect to status (feedback) of the system.

- The user wants clear feedback on the estimated time and progress of completion.
- The user may not be familiar with the complexity of the task.
- During the operation the user might decide to interrupt the operation because it might take too long for completion.
- The system should account for human needs and capabilities when deciding what aspects of System State to display and how to present them.
- The System State should be presented in a clear fashion, such that it does not confuse the user.

Problems: If the user does not know what is being done by the system, then the user will be frustrated.

Solution: To show to the user what application is doing and give a clear indication of the progression of the task being completed we need two things – collection of system state data and take initiative to present them in an informative ways.

Event component (Figure 14) gets the events from the user or the other systems. After getting events form the Event component, Decision about Initiative (DAI) consults with different models and take initiative to show the results in an informative way in the Presentation component.

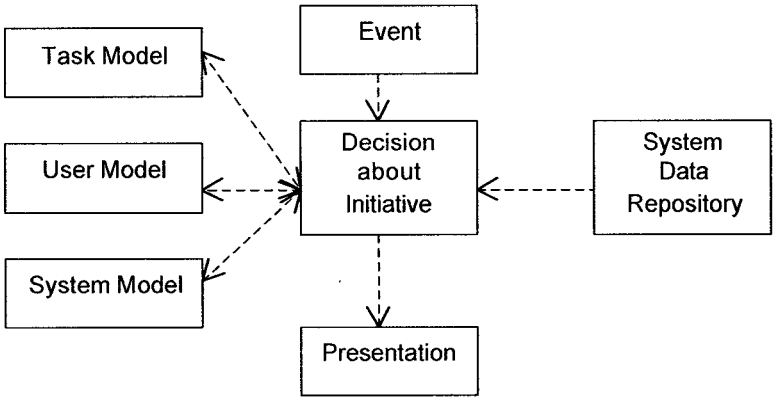

Figure 14: Provide user confidence pattern

4.9.1 Usage of such patterns within the MVC framework

All models and DIA will be the part of the core-functionality of the system. So, DAI can be changed without affecting Presentation and Data Repository components.

4.10 Working Data Visualization

Name of Pattern: Working Data Visualization (Presentation)

Context: Sometime users want to see data in different point of views to get insight idea about the data, so that they will have idea 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 in different viewpoint, so that they can edit in order to improve their document.

- Different user likes different viewpoint (mode).
- Each viewpoint (mode) should have related commands to manipulate data.

Problems: If user cannot see working data in different view mode to get better idea and if switching between views does not change related command of manipulation then user will be frustrated.

Solution: Data that is being viewed should be separated form the data view description (Figure 15), so that same data can be viewed in different ways according to the different view descriptions. Presentation gets the data and commands according to user selected view description.

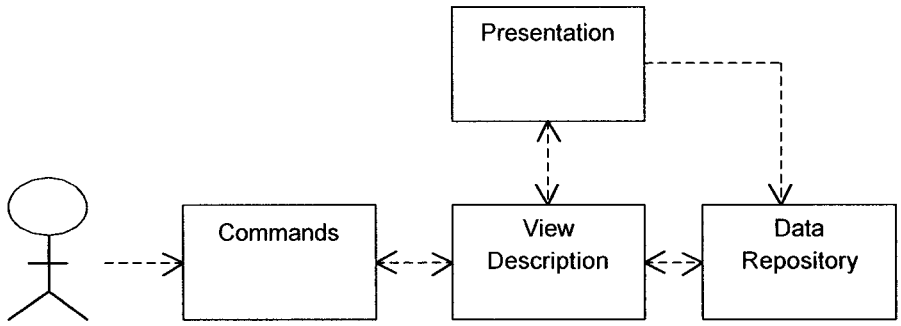

Figure 15: Working data representation in different ways

4.10.1 Usage of such patterns within the MVC framework

View description can be thought as core-functionality is separated for the Data and Presentation, so that each component can be modified without affecting each other.

5 Conclusion and future work

In this thesis, we presented an approach to improve the usability of software systems by means of software architecture. We first identified specific cause/effect relation between usability factors and internal attributes of software. We also formulated each cause/effect relationship as a scenario with characteristics of external attributes. For every scenario, we provided several architectural patterns that provide a solution to cause/effect relationship scenarios. We then showed how our patterns could be enhanced to MVC model (MVCforUnability).

This research effort can benefit software architecture designers and developers. They can use our MVCforUsability framework in three fashions:

1. The scenarios can serve as a checklist to show whether important usability features (external attributes) have been considered in the requirements by incorporating related external attributes of the software system.

2. The architecture patterns can help the designer to implement cause/effect relationship scenarios through MVCforUsability model to incorporate the usability in the system.

3. The relationship scenario enables a designer to determine what additional aspects of usability can be supported for minimal cost by analyzing internal and external attributes of the software.

This work is by no means complete. The following issues have to be explored:

- Validating and extending the list of cause/effect scenarios. The cause/effect relationship scenarios we proposed need to be validated in real world applications.
- Identifying the other internal attributes that can affect usability.

An Approach of validating the list of patterns:

Since our patterns are based on scenarios and each scenario is based on the relationship of internal and external attributes of the software system, so each scenario has specific set of requirements to be achieved through software pattern. Every pattern has a set of problems to be solved and a set of goals to be achieved. Designers and developers can use the use-case and scenarios as functional prototypes (simulations of the system) of the pattern and let users manipulate, critique, configure, annotate and enhance those prototypes. To validate each pattern designers and developers could follow the following steps:

1. Identify set of internal attributes and set of user requirements.
2. Apply use-case to the set of user requirements and analyze to find the ways to achieve goals.
3. Compare the ways of achieving goals with the design pattern and if necessary redistribute the internal attributes or components within the design pattern.
4. Since every pattern are proposed in a way that each functionality is separated from one another, so developer can built them separately and connect them through the use-case defined connection interface.

6 References

[1] Len Bass, Bonnnie E. John and Jesse Kates, *Achieving Usability Through Software Architecture*, March 2001, SEI, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

[2] Roger S. Pressman, *A Manager's Guide to Software Engineering*, McGraw-Hill, Inc., New York, NY, 1993.

[3] Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice*, Addison-Wesley, Reading, MA, 1998.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern Oriented Software Architecture: A System of Patterns*, John Wiley & Sons Ltd., West Sussex, England, 1996.

[5] Steve Burbeck, *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC),* ParcPlace Systems, Inc., 999 E. Arques Ave., Sunnyvale, California 94086-4593, UAS, 1992.

[6] Jeff Raskin, *The Humane Interface: New Directions for Designing Interactive Systems*, Addison-Wesley, 2000.

[7] William M. Newman, Michael G. Lamming, *Interactive System Design*, Addison Wesley, 1995.

[8] Rick Kazman, S. Jeromy Carroere, Steven G. Woods, *Toward a Discipline of Scenario-Based Architectural Engineering*, SEI, Carnegie Mellon University, Pittsburgh, USA, 2000.

[9] Chris Stary, Nikos Vidakis, *User Interface Design as Knowledge Management*, University of Linz, Department for Business Computing Communications Engineering, Linz, Austria.

[10] Laurence Nigay, Daniel salber, Simon Buckinghan Shum and Joelle Coutaz, *Teaching Trainee and Professional Designers to use the PAC-AMODEUS Software Architecture Modelling Technique*, Universite Joseph Fourier, France.

[11] Francis Jambon, *Error Recovery Representations in Interactive System Development*, LISI / ENSMA, BP 109 F-86960, Futuroscope cedex, France.

[12] Jiantao Pan, Philip Koopman and Daniel Siewiorek, *A Dimensionality Model Approach to Testing and Improving Software Robustness*, SEI, Carnegie Mellon University, Pittsburgh, USA, 1999.

[13] IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), IEEE Computer Soc., Dec. 10, 1990.

[14] William M. K. Trochim, Pattern Matching for construct validity, Cornell University, http://trochim.human.cornell.edu/kb/pmconval.htm, USA, 2000.

[15] Len Bass, Mark Klein, and Felix Bachmann, *Quality Attribute Design Primitives and the Attribute Driven Design Method*, SEI, Carnegie Mellon University, Pittsburgh, USA, 2001.

[16] Terry Winograd, *Architectures for Context*, Computer Science Department, Stanford University, 2001.

[17] Roy A. Maxion, Robert T. Olszewski, *Improving Software Robustness with Dependability Cases*, School of Computer Science, SEI, Carnegie Mellon University, Pittsburgh, USA, 1998.

[18] Len Bass, Mark Klein, Felix Bachmann, Quality Attribute Design Primitives, SEI, Carnegie Mellon University, Pittsburgh, USA, 2000.

[19] Len Bass, Mark Klein, Gabriel Moreno, *Applicability of General Scenarios to the Architecture Tradeoff Analysis Method*, SEI, Carnegie Mellon University, Pittsburgh, USA, 2001.

[20] Marc Evers, *A Case Study on Adaptability Problems of the Separation of User Interface and Application Semantics*, University of Twente, Dept. of Computer Science, Software Engineering Group, P.o. Box 217, 7500 AE Enschede, The Netherlands.

[21] Mohammad Donyaee, *Thesis: QUIM – A Model for Specifying and Measuring Quality in Use*, Dept. of Computer Science, Concordia University, Montreal, 2000.

[22] Dorin Sandu, *User Interface Patterns*, School of Computer Science, Carleton University, Ottawa, Ontario, Canada.