Chapter 5

Bridging Patterns

Published as: Bridging Patterns; an approach to bridge gaps between SE and HCI. Eelke Folmer, Martijn van Welie, Jan Bosch, accepted for the Journal of Information and Software Technology. 17 Jan 2005.

Abstract: Adding usability improving solutions during late stage development is to some extent restricted by the software architecture. However, few software engineers and human computer interaction engineers are aware of this important constraint and as a result avoidable rework is frequently necessary. In this paper we present a new type of pattern called a bridging pattern. Bridging patterns extend interaction design patterns by adding information on how to generally implement this pattern. Bridging patterns can be used for architectural analysis: when the generic implementation is known, software architects can assess what it means in their context and can decide whether they need to modify the software architecture to support these patterns. This may prevent part of the high costs incurred by adaptive maintenance activities once the system has been implemented and leads to architectures with better support for usability.

5.1 Introduction

A software product with poor usability is likely to fail in a highly competitive market; therefore software developing organizations are paying more and more attention to ensuring the usability of their software. Practice however shows that product quality (which includes usability among others) is not that high as it could be. Organizations spend a relative large amount of money and effort on fixing usability problems during late stage development (Pressman, 1992, Landauer, 1995). Some of these problems could have been detected and fixed much earlier. This avoidable rework leads to high costs and systems with less than optimal usability, because during the development different tradeoffs have to be made, for example between cost and quality.

This problem has been around for a couple of decades especially after software engineering (SE) and human computer interaction (HCI) became disciplines on their own. While both disciplines developed themselves, several gaps have appeared which are now receiving increased attention in research literature. Major gaps of understanding, both between suggested practice and how software is actually developed in industry, but also between the best practices of each of the fields have been identified (Carrol et al, 1994, Bass et al, 2001, Folmer and Bosch, 2002) (Folmer and Bosch, 2004). In addition, there are gaps in the fields of differing terminology, concepts, education, and methods. (Walenstein, 2003). Several problems and solutions have been identified to cover some of these gaps. (Constantine et al, 2003, Ferre, 2003, Milewski, 2003, Willshire, 2003).

Our approach to bridging one of these gaps is based upon the following observation: software engineers in general have few techniques available for predicting the quality attributes of a software system before the system itself is available. Therefore often systems are developed and the provided quality is measured only when the system is completed. If a usability problem is detected during late stage sometimes this is very expensive to fix because its solution may have a structural impact. We call this problem the retrofit problem (Folmer et al, 2003). The quintessential example that is always used to illustrate the retrofit problem is adding Undo. From experience it is learned that it can be very hard to implement undo in an application because it requires many system functions to be completely rewritten and it also prescribes certain components and relationships between these components. If a certain software architecture i.e. a fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution (IEEE, 1998) has already been implemented then changing or adding new components to this structure during late stage design is likely to affect many parts of the existing source code. Large parts of the code need to be rewritten and restructured which is of course very expensive.

Adding usability improving solutions during late stage development is to some extent restricted by the software architecture. However, few software engineers and human computer interaction engineers are aware of this important constraint and as a result avoidable rework is frequently necessary. User interface designers and software engineers have usually very different backgrounds resulting in a lack of mutual understanding of each others view on technical or design issues.

In this paper we present a new type of pattern that is called a 'bridging pattern'. This pattern 'bridges' one of the gaps between SE and HCI we identified during architectural design by describing a usability improving design solution that exhibits the retrofit problem. This pattern consists of a user interface part and an architecture/implementation part. When the architectural implications are known such a pattern can be used for architectural analysis e.g. deciding during architectural design whether this pattern needs to be accommodated by the architecture. This may prevent part of the high costs incurred by adaptive maintenance activities once the system has been implemented.

The remainder of this paper is organized as follows. In the next section, we present and define the concept of a bridging pattern. Section 3-6 present four examples of bridging patterns we have identified. Section 7 discusses some of the issues we encountered during the definition of bridging patterns. Section 8 discusses future work and the paper is concluded in section 9.

5.2 Bridging Patterns

Patterns and pattern languages for describing patterns are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience. Design patterns (Gamma et al 1995, Buschmann et al, 1996) are extensively used by software engineers for the actual design process as well as for communicating a design to others. Software patterns first became popular with the object-oriented Design Patterns book (Gamma et al 1995). Since then a pattern community has emerged that specifies patterns for all sorts of problem domains: architectural styles (Buschmann et al, 1996), object oriented frameworks (Coplien and Schmidt, 1995), domain models of businesses (Fowler, 1996), interaction patterns (Tidwell 1998, Welie and Trætteberg, 2000, PoInter, 2003) etc.

A lot of different types of patterns have been defined in the field of HCI; interaction patterns (Tidwell 1998, Welie and Trætteberg, 2000, PoInter, 2003) (undo), user interface patterns (Welie, 2003, Laakso 2004) (progress indicator), usability patterns (Brighton, 1998, Perzel and Kane 1999) (Model view controller), web design patterns (Duyne et al, 2002, Welie, 2003) (shopping cart) and workflow patterns (Workflow patterns). These patterns have a lot of similarities, some patterns are known under different names (or even the same name) in different pattern collections. The main thing they share is that they most commonly provide solutions to specific usability problems in interface and interaction design. A lot of work has been done in these fields in recent years and several pattern collections (Brighton, 1998, Common ground, 1999, Welie, 2003, PoInter, 2003) are freely accessible on the web. Our work focuses on these patterns which we from now on refer to as Interaction Design (ID) patterns.

Unlike the design patterns, ID patterns do not specify a specific design solution for example in terms of objects and classes. Most existing ID patterns collections refrain from providing or discussing implementation details. This is not surprising since most ID patterns are written by user interface designers that have little interest in the implementation part and usually lack the knowledge to define the implementation details. This is a problem since without knowing these implementation details it becomes very hard to assess whether such a pattern can be easily added during late stage design

Bridging patterns extend ID patterns by adding information on how to generally implement the pattern. When the generic implementation is known, software architects can assess what it means in their context and can decide whether they need modify the architecture to facilitate the use of this pattern. Some previous work has been done in this area: Trætteberg (Trætteberg, 2000) presents model fragments that can be used for UI patterns. Bass and John (Bass et al, 2001) identified scenarios that illustrate particular aspects of usability that are architecture-sensitive and suggest architectural patterns for implementing these scenarios. In (Folmer et al, 2003) we defined a framework that expresses the relationship between usability and software architecture consisting fifteen architecture sensitive usability patterns.

The problem with these approaches is that they are quite preliminary. For example Bass (Bass et al, 2001) gives a very high-level of the architecture needed for Undo. Basically he proposes a transaction manager component. Although that is not wrong, it is in our opinion too general to be of much practical help. (Folmer et al, 2003) state a more precise architectural description that talks about maintaining state information and the suggestion that undo may be implemented using the Command (Gamma et al 1995) pattern. However, while researching UNDO implementations, we found that of the many possibilities to capture state information, only one is commonly used. That is, the one were only state-changes against the original are stored in 'Command-like' objects. It requires a variant of the Command pattern where some more methods, classes and interfaces are used. The real difficulty arises when implementing the Command objects so that they work within a framework. Command objects have to deal with issues such as dynamic pointers and storing sufficient state to undo and redo the command. On top of that implementing 'selective undo' functionality puts even more constraints on the framework and the command objects. The question that arises from this reasoning is whether is it useful to talk about architectural sensitivity of some of the ID patterns without fully understanding the detailed issues when implementing this pattern.

We defined bridging patterns with the following purposes:

- Provide detailed implementation issues and solutions. This allows a software architect to assess the architectural impact of implementing such a pattern. Although applications do not become usable by using bridging patterns, they may help to become more aware of how to implement user interface things and what that could mean for the architecture of an application.

- Provide an instrument to facilitate communication across the HCI and SE boundaries. By discussing how a certain pattern is implemented and what its effect may be on the architecture mutual awareness of the restrictions that exist between SE and HCI can be raised. Even if the architecture consists of certain mechanisms (such as style sheets) software architects may become aware that these mechanisms may support certain usability features (such as multiple views). On the other hand UI engineers may become aware of the architectural impact of a particular design solution which allows them to understand that it is expensive to add such solutions during late stage.

In this paper we present four bridging patterns namely multi-level undo, multi channel access, wizard and single sign-on. This paper extends previous work (Welie and Trætteberg, 2000, Folmer et al, 2003) we have done on this topic. The Undo and Wizard patterns had already had been described by several ID pattern collections (Common ground, 1999, Welie and Trætteberg, 2000) though sometimes these patterns are described with different names (e.g. Tidwell undo = go one step back). Undo (Bass et al, 2001, Folmer et al, 2003), Wizard(Folmer et al, 2003) and Multichannel access (Folmer et al, 2003) have also been recognized as being architectural sensitive. Some of our bridging patterns such as Multi channel access and Single sign-on are not recognized in ID pattern collections however these patterns consist of elements such as providing multiple views (printable pages & personalize content (Duyne et al, 2002)) that are recognized in pattern and guideline literature. These patterns can be considered to be of a higher level of abstraction than ID patterns.

We analyzed the pattern's architectural impact and implementation by analyzing often-used applications and documentation. We merely extended existing ID patterns by adding an implementation part to the pattern. As identified by (Granlund et al, 2001) patterns are an effective way of capturing and transferring knowledge due to their consistent format and readability.

To describe our bridging patterns the following format is used:

Problem: Problems are related to the usage of the system and are relevant to the user of any other stakeholder that is interested in usability.

Use when: a situation (in terms of the tasks, the users and the context of use) giving rise to a usability problem. The use when extends the plain problem-solutions dichotomy by describing situations in which the problems occur.

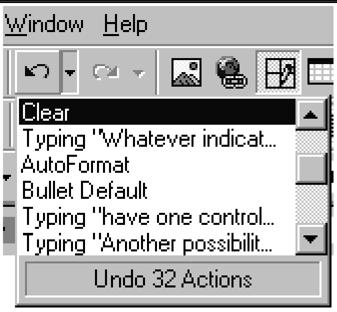
Solution: a proven solution to the problem. However a solution describes only the core of the problem but other patterns may be needed to solve sub problems.

Why: How and why the pattern actually works including an analysis how it may affect certain attributes of usability. The rationale (why) should provide a reasonable argumentation for the specified impact on usability when the pattern is applied. The why should describe which usability aspects should have been improved or which other aspects have to suffer (Welie and Trætteberg, 2000).

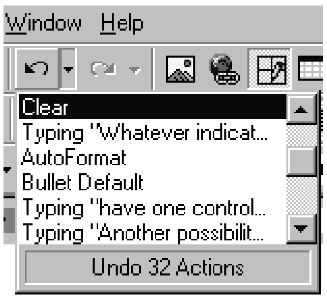
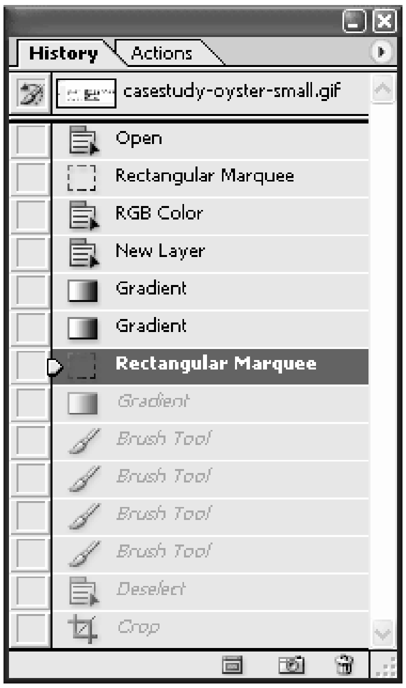
Examples: The example shows how the pattern has been successfully applied in a real life system. This is often accompanied by a screenshot and a short description.

Architectural implications: An analysis of the structural impact of the pattern and which responsibilities may need to be fulfilled by the architecture. Often these responsibilities can be fulfilled by the use of design patterns (Gamma et al 1995) or the use of architecture styles and patterns (Buschmann et al, 1996).
Implementation: Specific implementation details in terms of classes and objects or either in terms of technologies or techniques that should be used. The implementation part gives a basic framework or architecture and points to the main issues while implementing.

5.3 Multi Level Undo

Table 32: Multi Level Undo	
	 Window Help Clear Typing "Whatever indicat... AutoFormat Bullet Default Typing "have one control... Typing "Another possibilit... Undo 32 Actions **Figure 34: Undo in Microsoft Word**
Problem	Users do actions they later want reverse because they realized they made a mistake or because they changed their mind.
Use when	You are designing a desktop or web-based application where users can manage information or create new artifacts. Typically, such systems include editors, financial systems, graphical drawing packages, or development environments. Such systems deal mostly with their own data and produce only few non-reversible side-effects, like sending of an email within an email application. Undo is not suitable for systems where the majority of actions is not reversible, for example, workflow management systems or transaction systems in general. Both novice and expert users may want to reverse their actions, either because of mistakes or changes in intention. Expert users may want to use the history of their actions for more specific manipulation of the data in the application. For example, in a graphical modeling application, users may want to undo work on some specific object while keeping later work done on other objects.
Solution	**Maintain a list of user actions and allow users to reverse selected actions.** Each 'action' the user does is recorded and added to a list. This list then becomes the 'history of user actions' and users can reverse actions from the last done action to the first one recorded. This is also called a Linear Multi-level Undo. **Interacting with the history**

	There are two variations on how to show the history of actions to the users. First there is the standard 'office-like' way where the 'Edit' menu contains both 'Undo' and 'Redo' functions with their keyboard shortcuts. Often there is also a widget in the toolbar that can show the last items in the history. By dragging the selection in the list, actions can be undone. A second variant is to work with primarily with the history list itself and moving a slider or scrollbar to move back in history and undo actions. Photoshop uses such a variant. **Displaying actions** Actions in the history are usually displayed using a text label such as 'Create circle', 'Typing',' New contact'. Such labels only name the function and not the object the functions work on. In some applications it may be better to include the object and the parameters as well, for example 'Change-color Circle12 to Red'. **Granularity of actions** When designing Undo it is important to determine the desired granularity of actions. For example, it is usually not desired to record each key press in a text editor as an action. Instead, typing a word is used as a unit of action. Designers need to determine what unit of action is appropriate in the application. **Non-reversible actions** Although most actions in the application may be reversible, it is very likely that some actions will not be reversible. For example, printing, saving, doing a payment, or downloading an object. For actions that are non-reversible and 'negative' of nature (like paying or destroying something), need to show the user a Warning Message and not add the action to the history. **Selective undo** In some cases, it can be meaningful to allow single actions from the history to be deleted. This is the case when a certain 'episode' of work must be deleted or undone while keeping work that has been done later on. Selective undo is conceptually much more difficult than linear undo since there is a notion of 'dependency between actions' that determines the consequences of undoing a particular action. For example, if a 'create circle' action is undone at some point in the history, subsequent actions in the history working on that object loose their meaning and must be deleted. There are many semantic issues with selective undo, see (Berlage, 1994) for more information on selective undo. **Object-based Undo** Object-based Undo can sometimes be considered as an alternative to Selective Undo. With Object-based Undo, each object has its own action history. Upon selecting the object, the users can undo actions done on the object. Naturally, this requires the application to have a clear concept of an 'object' and is therefore not applicable for bitmap editors. See (Zhou and Imamiya, 1997) for more on Object-based Undo. **Multi-user undo** If the application is a multi-user application and uses undo, the application must distinguish between local actions and global actions. That leads to multiple histories and requires special semantics for what happens when undoing actions. See (Abowd and Dix, 1992, Ressel and Gunzenhouser, 1999, Sun, 2000) for more on multi-user undo issues.
Why	Offering the possibility to always undo actions gives users a comforting feeling. It helps the users feel that they are in control of the interaction rather than the other way around. They can explore, make mistakes and easily go some steps back, which facilitates learning the application's functionality. It also often eliminates the need for annoying warning messages since most actions will not be permanent

More Examples	 Window Help Clear Typing "Whatever indicat... AutoFormat Bullet Default Typing "have one control... Typing "Another possibilit... Undo 32 Actions **Figure 35: Undo in Word** As in all MS Office applications, in Word 2000 the users can see the history of their actions and undo one or more of them. The actions are briefly described and the users can select a range of actions to be undone. After selecting undo, users can even redo the actions.	 History Actions casestudy-oyster-small.gif Open Rectangular Marquee RGB Color New Layer Gradient Gradient Rectangular Marquee Gradient Brush Tool Brush Tool Brush Tool Brush Tool Deselect Crop **Figure 36: Undo in Photoshop** In Photoshop a selective undo is also possible. By moving the slider, users can do the normal multi-level undo but they can also drag an action into the trashcan and thereby do a selective undo. Operations that depended on that action are automatically deleted as well of they are not relevant anymore.
Architectural Considerations	There are basically two possible approaches to implementing Undo. The first is to capture the entire state of the system after each user action. The second is to capture only relative changes to the system's state. The first option is obviously needlessly expensive in terms of memory usage and the second option is therefore the one that is commonly used. Since changes are the result of an action, the implementation is based on using Command objects that are then put on a stack. Each specific command is a specialized instance of an abstract class Command. Consequently, the entire user-accessible functionality of the application must be written using Command objects. When introducing Undo in an application that does not already use Command objects, it can mean that several hundred Command objects must be written. Therefore, introducing Undo is considered to have a **high** impact on the software architecture.	
Implementation	Most implementations of multi-level undo are based on the Command (Gamma et al 1995) pattern. When using the Command pattern, most functionality is encapsulated in Command objects rather than in other controlling classes. The idea is to have a base class that defines a method to "do" a command, and another method to "undo" a command. Then, for each command, you derive from the command base class and fill in the code for the do and undo methods. The "do" method is expected to store any information needed to "undo" the command. For example, the command to delete an item would remember the content of the item being deleted. The following class diagram shows the basic Command pattern structure:	

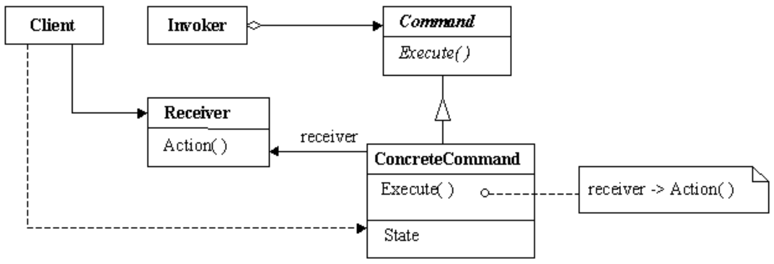


Figure 37: Command Pattern

In order to create a multi-level undo, a Command Stack is introduced. When a new command is created, its 'Do' function is called and the object is added to the top of the stack if the command was successful. When undoing commands, the 'Undo' function of the command object at the top of the stack is called and the pointer to the current command is set back.

When you want to redo a command, you increment the stack pointer and call the "do" method of the object at the stack pointer. Note that the act of pushing a new command to the command stack truncates the stack at that point, discarding all of the command objects after the current top-of-stack stack entry. Redoing and undoing just moves the stack pointer up and down the stack.

Participants

The classes and/or objects participating in this pattern are:

- Command (Command), declares an interface for executing an operation.
- ConcreteCommand (CalculatorCommand) defines a binding between a Receiver object and an action and implements Execute by invoking the corresponding operation(s) on Receiver.
- Client (CommandApp) creates a ConcreteCommand object and sets its receiver.
- Invoker (User) asks the command to carry out the request.
- Receiver (Calculator) knows how to perform the operations associated with carrying out the request.

A practical Undo framework

Although Multi-level Undo is based on the Command pattern, in practice a more elaborate framework is needed. In some toolkits such as Java Swing, such a framework is already present. The following picture shows the main components:

This framework shows that there are additional methods necessary for implementing Multi-level Undo. First of all there are extra methods in the Command class that allow the UI objects to display names for the objects. Then you can see that the stack has a limited size which can be configured. This is often necessary since there can be a lot of memory used by Command objects because of the information they need to remember. Then there is a 'Compound' command class. In practice it shows that what is one command from a user's perspective are actually several commands at the implementation level. Compound commands allow such groupings of command objects.

Dynamic Pointers

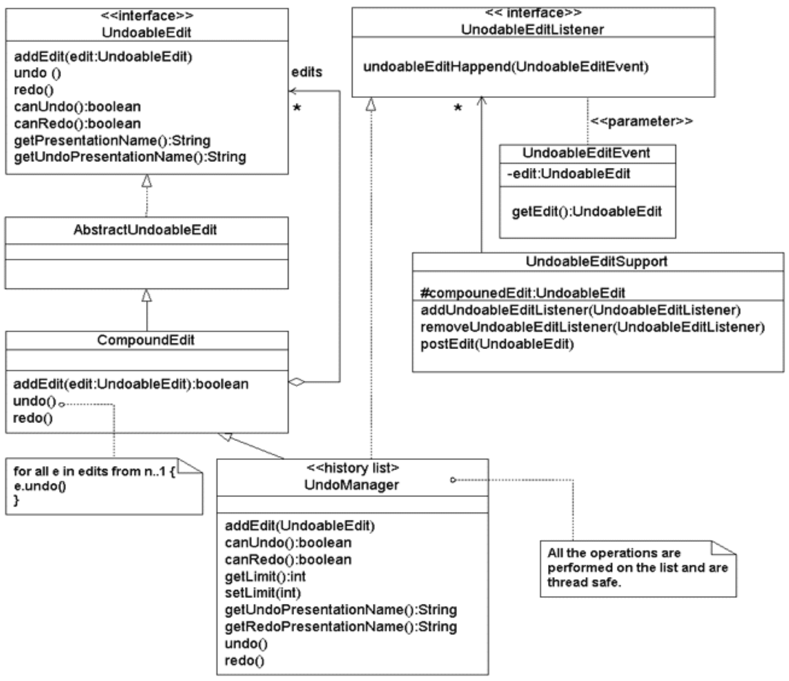
When implementing Command objects, there is an issue of using pointers. When a command is created all pointers will usually be valid but at the time the command is undone, there is no hard guarantee that the pointers are still valid. Anything may have happened to those objects in the mean time. Depending on the actual application it may be necessary to use 'dynamic pointers' that allow the command object to retrieve a valid pointer to the right object.

Storing state information

Since the Command object must be able to undo the operation, enough information must be captured to be able to do so. Usually, the command objects stores aspects of the objects it works on. Alternatively, the memento (Gamma et al 1995) pattern can be used to anonymously store the state of an object and to set a stored state back.

Selective undo

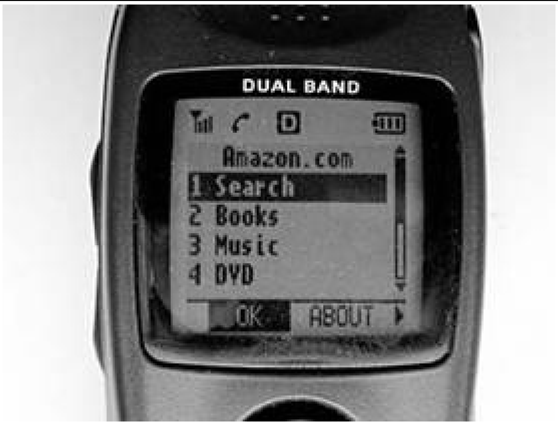
If you want to support selective undo where an arbitrary command in the stack can be undone, you must deal with dependencies between command objects. The most common way to implement this is to add a method 'isPossible()' to command objects. When a command is undone, a check is done whether the next command is still possible. If it is the object can stay on the stack and a check is done for the subsequent command object. If it is not possible, the command is removed from the stack and the subsequent command is checked.

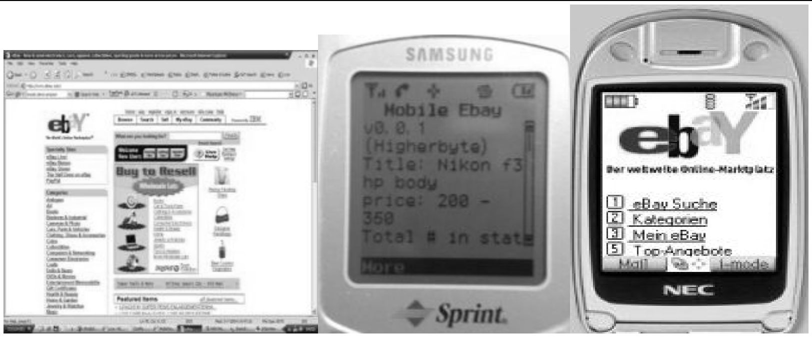


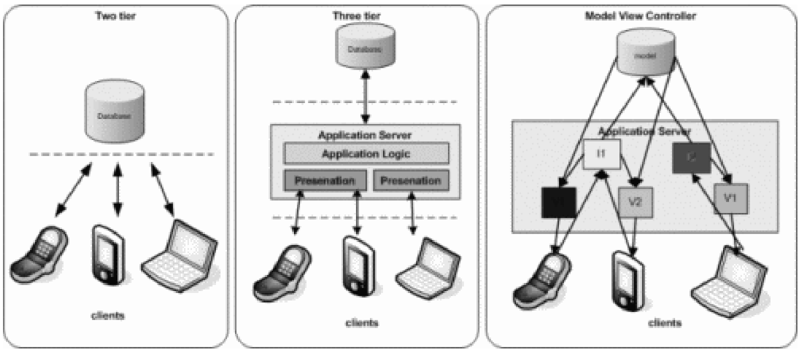
<<interface>>
UndoableEdit
addEdit(edit:UndoableEdit)
undo ()
redo()
canUndo():boolean
canRedo():boolean
getPresentationName():String
getUndoPresentationName():String
edits
*
<< interface>>
UnodableEditListener
undoableEditHappend(UndoableEditEvent)
*
<<parameter>>
UndoableEditEvent
-edit:UndoableEdit
getEdit():UndoableEdit
AbstractUndoableEdit
UndoableEditSupport
#compounedEdit:UndoableEdit
addUndoableEditListener(UndoableEditListener)
removeUndoableEditListener(UndoableEditListener)
postEdit(UndoableEdit)
CompoundEdit
addEdit(edit:UndoableEdit):boolean
undo()
redo()
for all e in edits from n..1 {
e.undo()
}
<<history list>
UndoManager
addEdit(UndoableEdit)
canUndo():boolean
canRedo():boolean
getLimit():int
setLimit(int)
getUndoPresentationName():String
getRedoPresentationName():String
undo()
redo()
All the operations are performed on the list and are thread safe.

Figure 38: Undo Framework

5.4 Multi Channel Access

Table 33: Multi Channel Access	
	 DUAL BAND Amazon.com 1 Search 2 Books 3 Music 4 DVD OK ABOUT **Figure 39: Amazon Accessible through a WAP Browser**
Problem	A user wants or requires access to a system using different devices (mobile phone, desktop, PDA).
Use when	You are designing a web based system (such as an e-commerce system) that targets many users. To increase the number of potential users/customers the accessibility of the system is increased by supporting multiple devices. The need for a device is either determined by the users (for example a disabled person with speech input device) or either by the user context (for example a mobile phone) or by a combination of these. An application that has been designed for the Web will contain lists of items that would fit in a desktop screen without problems but that we are unavailable to display completely on the very small screen of a mobile phone. Interaction on a mobile phone is also limited. Therefore the same software functionality is required to be manipulated and presented using different human-computer controls and different interface styles for different user preferences, needs or disabilities. Devices are different with regard to input device (e.g. mouse, stylus, keyboard, voice) but also with regard to output device (screen size, resolution, screen colors, audio etc). In addition devices may pose constraints because they are limited concerning hardware (memory, CPU power), Operating systems (multi/single threaded), communication types (asynchronous/synchronous) and software (languages/ character sets/ browsers) posing constraints on the ability to present functionality in the same way as for example on a desktop computer.
Solution	**Provide a mechanism that provides multiple channels that are specialized to support different devices.** Multi channeling defines the user's accessibility to a system through more than one channel. Each channel specifies a certain presentation and or control set for a device or group of devices. Different input devices often also prescribe certain interface components (such as projecting a keyboard on a touch screen). How this information is presented and manipulated may be different for each device but the accessibility (e.g. access to the system) is the same for each user.
Why	Accessibility may increase satisfaction by allowing the use of the system adapted to the users (user context, device, disability etc). Providing device specific views may aid visual consistency and functional consistency

	hence contributing to learnability. Having device-specific views available at any time will contribute to error prevention and minimize cognitive load.
More Examples	 Mobile Ebay v0.0.1 (Higherbyte) Title: Nikon f3 hp body price: 200 - 360 Total # in state More SAMSUNG Sprint. ebay Der weltweite Online-Marktplatz 1 eBay Suche 2 Kategorien 3 Mein eBay 5 Top-Angebote Mail i-mode NEC **Figure 40: Ebay on PC, WAP and I-Mode Phone** Ebay.com can be accessed through conventional channels such as a desktop computer, but also through more advanced channels such as an I-mode/WAP phone or a PDA. Although the how the information is presented and manipulated is different for each device, the accessibility is the same for each channel which allows users to fulfill their goal of use (e.g. buy a specific object on eBay)
Architectural Considerations	The architecture must be constructed in such a way that new channels can be easily added and maintained. There are several architecture styles (Buschmann et al, 1996) that fulfill these responsibilities. **2-tier architecture** The classical two-tier architecture divides the software system into server and clients. The client application contains business logic as well as the complete interaction. There is no separation of functionality and interaction. In traditional monolithic / two tier applications when adding new channels business functionality and interaction logic needs to be duplicated for each channel. This solution works fine if the number of devices that need to be supported is small. **3-tier architecture** A more elegant solution is to use a 3-tier architecture. The 3-tier architecture separates the application into two tiers by separating the application logic from the user interface. In most cases, the application-logic part of the software on the server side is specially designed for a specific type of user interface. In this case the presentation is encapsulated but the interaction is not separated from business logic. When adding new channels part of the business logic and interaction in the second layers need to be duplicated. 2 and 3 tiered architecture are all examples of the architectural pattern layers (Gamma et al 1995). **Model view controller** One step further is to decouple the interaction from the business logic. The MVC pattern (Buschmann et al, 1996) is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller. • The view manages the output device (PDA screen, voice) that is allocated to its application. • The controller interprets input device (mouse, stylus or keyboard) 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 model component needs to notify the view component when the model is updated, so that the display can be redrawn.

	This pattern decouples changes to how data are manipulated from how they are displayed or stored, while unifying the code in each component. This leads to greater flexibility. There is a clearly defined separation between components of a program -- problems in each domain can be solved independently. New views and controllers (and hence devices) can be easily added without affecting the rest of the application.  Two tier Database clients Three tier Database Application Server Application Logic Presenation Presenation clients Model View Controller model Application Server f1 V2 V1 clients **Figure 41: 2T/3T and MVC Architectures** Supporting Multichannel access by MVC or n-tier architectures is considered to have a **high** impact on the software architecture. When introducing Multichannel Access in an application that does not already use layers or MVC (such as traditional monolithic applications), it means that a large part of existing functionality must be rewritten and reallocated. Responsibilities which are dispersed in a monolithic application need either be allocated to specific layers and to specific entities (client/server) or to specific controller and view components on specific devices.
Implemen-tation	In order to provide Multichannelling several mechanisms and techniques are required. **Being able to describe data independent from how it is displayed on a device.** This is often done using XML (Extensible Markup Language). XML allows designers to create their own customized tags, enabling the definition, transmission, validation, and interpretation of data between applications and between organizations. **A mechanism to convert device independent data to device specific presentation.** XSL Transformations (XSLT) is a language for transforming XML documents into other another form. This could be another XML document, or a document in a different format altogether, such as PDF, HTML, or even Braille. XSLT style sheets work as a series of templates which produce the desired formatting effect each time a given element is encountered. One of the most common uses of XSLT is to apply presentational markup to a document based on rules relating to the structural markup. Using XSLT XML objects can be transformed to a specific format suitable for a specific channel (for example WML or HTML). **A mechanism to convert device independent data to device specific interaction and views.** Generating the presentation for clients with different interaction models and flows of control (for example WAP vs. Desktop) requires very different transformations. Supporting such transformations increases development and runtime costs. A more efficient solution is to define client specific controllers and views. This can be done using Java Server Pages (JSP) and or Struts. The Struts framework is a flexible control layer based on standard technologies like Java Servlets, JavaBeans, Resource Bundles, and XML, as well as various Jakarta Commons packages. Struts encourages application architectures based on the Model 2 approach (also known as the JSP Model 2 architecture), a variation of the classic MVC design paradigm. Several View and Controller components can be defined for different devices, while all target devices have a common Model component, which encapsulates business logic and

data and is implemented as JavaBeans. To map this model on the persistent data some data abstraction layer is needed. Using Struts and JSP and or XSLT the application can be converted to device specific applications (views + controller) each of which contains one or more specialized JSP files. For example a dialog can be split into multiple specialized JSP's for devices with small screens. These specialized JSP's support the markup languages that their particular devices require, such as HTML or WML.

See (Seshadri, 1999, Sun, 2004) for more on implementing Multichannel access with Java Server Pages and XML. Other types of implementations using MVC are also possible for example model 2X approach (Mercay and Gilbert, 2002), .NET MVC application framework (Stuart et al,, asp.net).

A mechanism to assign devices to specific channels.

Certain device profiles can be defined which describe individual devices detailed capabilities. A persistent storage device such as a database should record these definitions for different types of client devices such as mobile phones, PDA's, and desktop clients. Alternatively an online profile repository (w3 uaprof) may be used.

Steps 1+2+4 lead to a typical 3 tier architecture (Model 1)

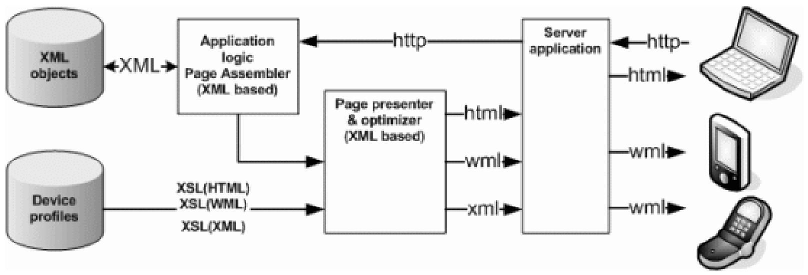


Figure 42: Model 1 Architecture

Steps 1+2+3+4 lead to typical MVC architectures (Model 2)

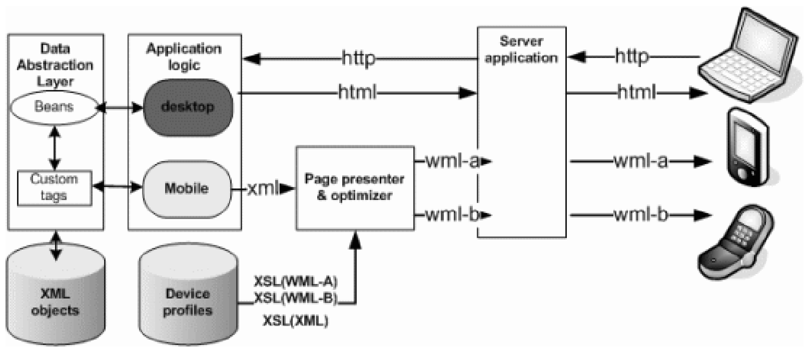
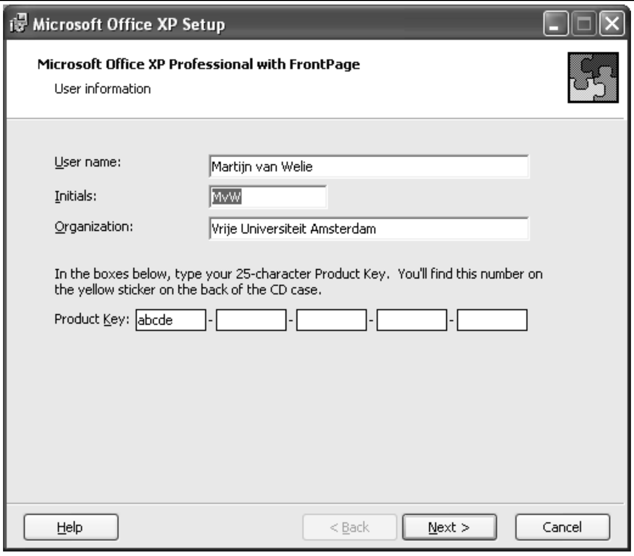
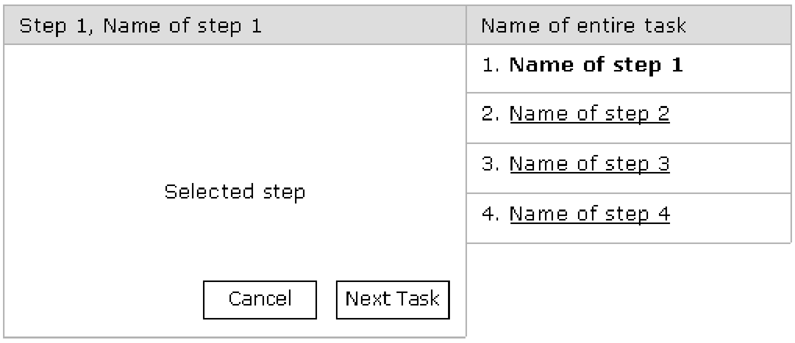
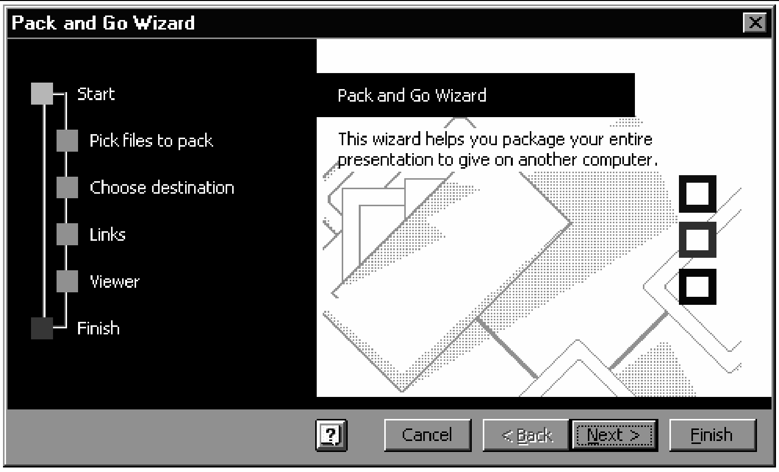


Figure 43: Model 2 Architecture

5.5 Wizard

Table 34: Wizard	
	 Microsoft Office XP Setup. Microsoft Office XP Professional with FrontPage. User information. User name: Martijn van Welie. Initials: MvW. Organization: Vrije Universiteit Amsterdam. In the boxes below, type your 25-character Product Key. You'll find this number on the yellow sticker on the back of the CD case. Product Key: abcde - - - - **Figure 44: a Typical Wizard**
Problem	The user wants to achieve a single goal but several decisions need to be made before the goal can be achieved completely, which may not be known to the user.
Use when	A non-expert user needs to perform an infrequent complex task consisting of several subtasks where decisions need to be made in each subtask. The number of subtasks must be small e.g. typically between ~3 and ~10. The user wants to reach the overall goal but may not be familiar or interested in the steps that need to be performed. The task can 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 several steps need to be taken but the exact steps required may vary because of decisions made in previous steps. In some cases, a Wizard may act as a 'macro'.
Solution	Take the user through the entire task one step at the time. Let the user step through the tasks and show which steps exist and which have been completed. When the complex task is started, the user is informed about the goal that will be achieved and the fact that several decisions are needed. The user can go to the next task by using a navigation widget (for example a button or some other form of mechanism). If the user cannot start the next task before completing the current one, feedback is provided indicating the user cannot proceed before completion (for example by disabling a navigation widget). The user is also able to revise a decision by navigating back to a previous task. The users are given feedback about the purpose of each task and the users can see at all times where they are in the sequence and which steps are part of the sequence. When the complex task is completed, feedback is provided to show the user that the tasks have been completed and optionally results have been processed. If relevant, users that know the default options can immediately use a shortcut that allows all the steps to be done in one action. At any point in the sequence it is possible to abort the task by choosing the visible exit.

	 Step 1, Name of step 1 Name of entire task 1. Name of step 1 2. Name of step 2 3. Name of step 3 4. Name of step 4 Selected step Cancel Next Task **Figure 45: Wire Frame Wizard**
Why	The navigation buttons suggest the users that they are navigating a path with steps. Each task is presented in a consistent fashion enforcing the idea that several steps are taken. The task sequence informs the user at once which steps will need to be taken and where the user currently is. The learnability and memorability of the task are improved but it may have a negative effect of the performance time of the task. When users are forced to follow the order of tasks, users are less likely to miss important things and will hence make fewer errors.
More Examples	 Pack and Go Wizard Start Pick files to pack Choose destination Links Viewer Finish Pack and Go Wizard Cancel < Back Next > Finish This wizard helps you package your entire presentation to give on another computer. **Figure 46: Pack-and-go Wizard in PowerPoint** The Pack-and-Go Wizard in Microsoft PowerPoint is one of the few wizards that also give an overview of the tasks and where the users are in the list.
Architectural Considerations	In order to implement Wizards, a small framework is needed. Once such a Wizard framework has been implemented it is fairly easy to create individual Wizards The architectural impact is **medium** for introducing a Wizard framework in the application and **low** for creating individual wizards when a framework is in place. However a complicating factor may be that Command (Gamma et al 1995) objects maybe needed in that case the architecture impact is **high**.
Implemen-tation	Technically speaking a wizard is a set of 'property pages' (or 'forms') that are filled in one by one. Basic ingredients of a Wizard Framework: • Wizard component (manages pages and navigation/help buttons, does the final execution of Commands). Usually this component also determines what the logic is between steps. • WizardStep containing the 'form' often implemented as a 'property page'. Must

remember the input values. Defines title, subtitle, image and form.

- WizardListener for events such as "next", "finish", "exit".
- Model, the WizardStep works ultimately on a model which will be modified once the Wizard is completed.

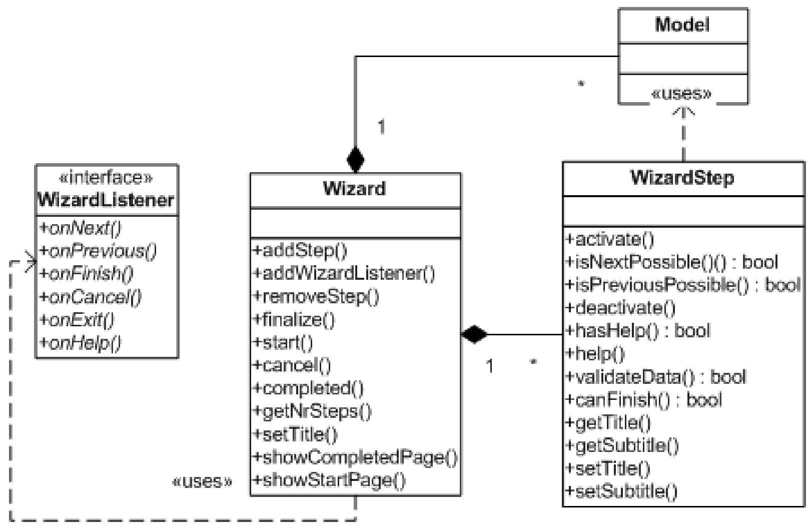


Figure 47: UML Model of a Wizard

The complete transaction is done after the "finish" event. If the application uses Multi-level Undo, the finish code creates the command objects and executes them. After the finish event has been handled successfully, the 'completion page' is shown to give users feedback.

Opening and finishing page

In the new wizard's Welcome page design, a general explanation of the wizard's task or purpose is stated. A list or overview of the main or specific tasks can be included in the Welcome page. The Completion page provides closure for the wizard and can contain either a general description or a specific list of what was accomplished in the wizard. The Completion page can also point the user to related tasks to be accomplished following wizard completion.

Navigating from step to step

In the case of pure linear wizards, the Wizard class may receive an event that the 'next' button has been pressed. The wizard class must check whether the form is valid, and if so, show the new step while storing the information of the previous step. The Wizard class asks the WizardStep whether or not the 'next'/previous button can be enabled. The Wizard class must als check whether the 'finish' button can be enabled.

Dynamic paths

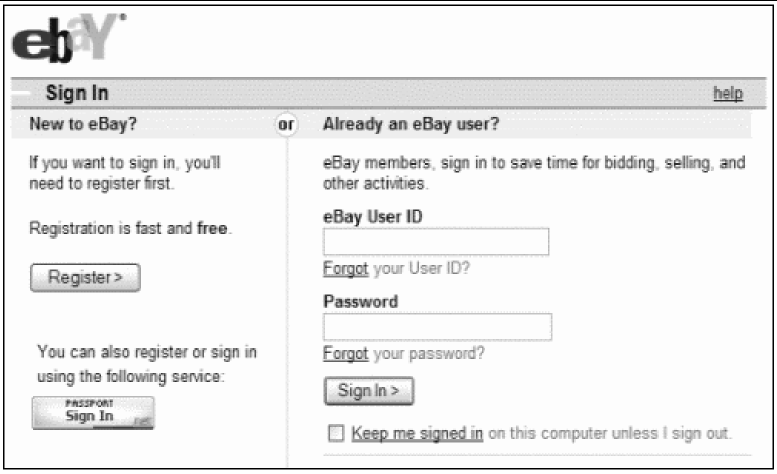
In complex wizards, the path users take using the wizards is not strictly linear. For example, many installation Wizards contain a step where user can choose between 'standard installation' and 'custom installation'. In such cases, the path has branching points and a Wizard navigator class is needed to manage what the next step needs to be.

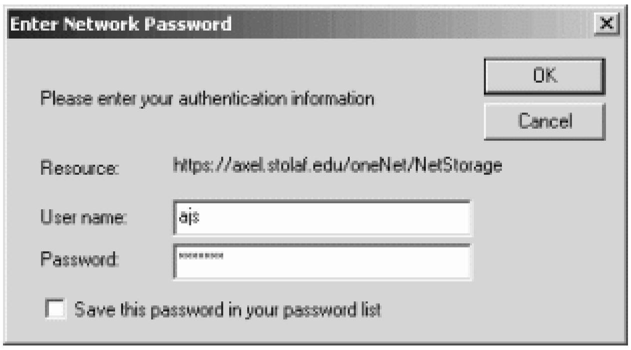
Execution and cancellation

While the user is going through all the steps, the WizardStep's simply remembers the entered data. When the users selects 'finish', the Wizard class has the responsibility to really execute the thing the wizard does. It uses the information collected in the WizardStep's and creates the necessary Command objects. If Multi-level Undo is used, the commands are executed and added to the stack. If the user wants to cancel during any of the wizard steps before the finalizing step, all data collected is simply destroyed and the Wizard exits. In some cases it may be necessary to already execute

	steps so that the next step can be performed. In such cases, Command objects are needed so that the effects can be cancelled when the user cancels the wizard.

5.6 Single Sign On

Table 35: Single Sign On	
	 Sign In help New to eBay? or Already an eBay user? If you want to sign in, you'll need to register first. Registration is fast and **free**. Register > You can also register or sign in using the following service: PASSPORT Sign In eBay members, sign in to save time for bidding, selling, and other activities. **eBay User ID** Forgot your User ID? **Password** Forgot your password? Sign In > ☐ Keep me signed in on this computer unless I sign out. **Figure 48: .NET Passport Providing SSO Capability**
Problem	The user has restricted access to different secure systems for example customer information systems or decision support systems by for example means of a browser. The user is forced to logon to each different system. The users who are allowed to access each system will get pretty tired and probably makes errors constantly logging on and off for each different system they have access to.
Use when	When you have a collection of (independent) secure systems which allows access to the same groups of users with different passwords.
Solution	**Provide a mechanism for users to authenticate themselves to the system (or system domain) only once.** The user can authenticate and authorize him/her by a single action to all systems where he/she has access permission. After that the identity and session of the users is preserved across heterogeneous servers/systems as if all servers/systems were fully integrated, which avoids having to enter multiple passwords.
Why	If users don't have to remember different passwords but only one, end user experience is simplified. The possibility of sign-on operations failing is also reduced and therefore less failed logon attempts can be observed. Security is improved through the reduced need for a user to handle and remember multiple sets of authentication information (However having only one password for all domains is less secure incase the user loses his password). When users have to provide a password only once, routine performance is sped up including reducing the possibility of such sign-on operations failing. Reduction in the time taken, and improved response, by system administrators in adding and removing users to the system or modifying their access rights. Improved security through the enhanced ability of system administrators to maintain the integrity of user account configuration including the ability to inhibit or remove an individual user's access to all system resources in a coordinated and consistent manner.

More Examples	**Figure 49: .NET Password** Microsoft .NET Passport has become one of the largest online authentication systems in the world, with more than 200 million accounts performing more than 3.5 billion authentications each month. Passport participating sites include NASDAQ, McAfee, Expedia.com, eBay, Cannon, Groove, Starbucks, MSN Hotmail, MSN Messenger, and many more.  Enter Network Password / Please enter your authentication information / Resource: https://axel.stolaf.edu/oneNet/NetStorage / User name: ajs / Password: ******** / Save this password in your password list **Figure 50: Windows 2000 SSO** Microsoft Windows 2000 operating system provides an integrated, comprehensive and easy-to-use SSO capability. SSO is provided natively in Windows 2000 by means of the built-in Kerberos and Secure Sockets Layer protocols, which also can provide standards-based SSO within mixed networks.
Architectural Considerations	There are several solutions to providing SSO capabilities and some of those have architectural implications. In general the system- or software architecture must fulfill several responsibilities: **Encapsulation of the underlying security infrastructure** Authentication and authorization are distributed and are implemented by all parts of the system or by different systems. Authentication and or authorization mechanisms on each system should be moved to a single SSO service, component or trusted 3rd party dedicated server. Other systems and or parts of the system should be released from this responsibility. The SSO server/component/service can act as a wrapper around the existing security infrastructure to provide a single interface that exports security features such as authentication and or authorization. Either some encapsulating layer or some redirection service to the SSO service needs to be established. **Integration of user records** User records (which contains authentication and authorization details) of different systems should be merged in to one centralized system to ease the maintenance of these records. In case of a trusted 3rd party trusted service some local access control list needs to be created which maps the passports of the 3rd party trusted service to authorization profiles. Providing SSO has a **high** impact on the software architecture.
Implemen-tation	The most important aspect of providing SSO is encapsulation of existing security functionality and providing one interface for authentication and or authorization to the user. In this implementation we consider typical client-server systems since

providing SSO capabilities to standalone systems

In most cases encapsulation is done on the server side. However there are also solutions where some sort of form filler agent in installed in a clients browsers that provides SSO capabilities but we leave this option out of our discussion since this type of solution does not integrate user records. Encapsulation of authorization and authentication can either be done by encapsulating each system or all systems.

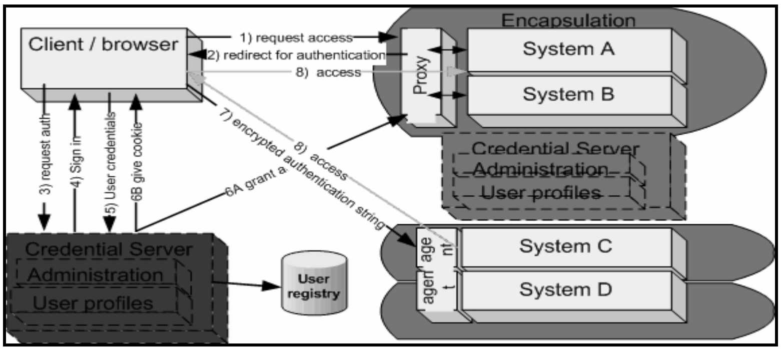


Figure 51: Encapsulation on Each System.

One can put a wrapper around each system that needs SSO. This can be done by installing an agent/ service/component on each system that deals with the authentication and or authorization. This agent should redirect the login attempt to a centralize credential server. The user logs in against a central authentication and or authorization server, the server then issues a session for the user's browser or system (for example IP nr) to SSO into all systems. In case the central server only provides authentication (for example with a trusted 3rd party identify provider), authorization still needs to be done on the encapsulated system. Authorization should then be done on a local centralized authorization server which manages a local access control list with user ID's from the trusted 3rd party. If authorization succeeds a valid session should be provided into all systems.

Encapsulation around all systems.

One can put a wrapper around all systems that needs SSO. This is especially useful for web based applications. A proxy is used to capture all http requests to the systems. Only requests with a valid session credential issued by the centralized authorization server are permitted. This design avoids any installations on the application servers. In this case the central authorization server provides authentication as well as authorization.

Location of centralized server

The centralized authentication / authorization server can be a part of a cluster of SSO systems or can be a trusted 3rd party SSO provider. In the case of a trusted 3rd party it is often not possible for system administrators to manage the user's credentials. A trusted 3r party often only provides an authentication mechanism just like a passport. Authorization still needs to be done on the local server. Some mapping between the user's passport and a local access control list needs to be established.

Microsoft passport

Microsoft .NET Passport provides an authentication service. This means that the service will determine whether users are who they claim to be, not whether they are allowed (authorization) to access resources. Participating sites can implement their own access control functionality to manage authorization after users have been authenticated (.net microsoft).

Users may obtain a .NET Passport account by providing two pieces of information: a unique e-mail address and password. This allows the user to take advantage of the benefits of the .NET Passport service, while maintaining a degree of anonymity. The .NET Passport service will verify a user's claimed identity only in the sense that he or she is the owner of the .NET Passport.

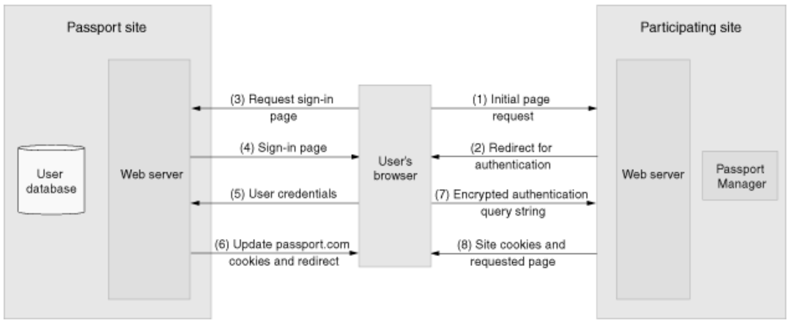


Figure 52: Overview of Passport Mechanism

Implementing the MS Passport begins by including the Passport Manager object as a server-side object on any webpage that requires authentication (Rauschenberger , 2001).

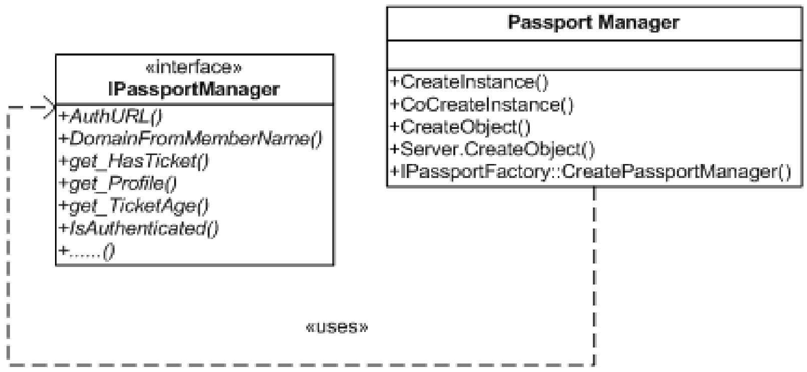


Figure 53: UML Model Passport Manager

The Passport Manager object interacts with Microsoft® Internet Information Services (IIS) and active server pages (ASP) to automatically handle .NET Passport cookie reads and writes, and provides information to ASP pages by calling various Passport Manager application programming interface (API) methods and properties. This object checks whether Passport has already authenticated the user for a particular site by looking for an authentication ticket in the user's cookies. If the ticket exists and is "fresh," the site treats the user as authenticated, and can access the user's profile information. Microsoft provides passport managers API's for languages such as C++, C# and VBScript (Microsoft, 2004). Once this object is created, each page can make a call using the IsAuthenticated method to determine if the user has signed into Passport. If IsAuthenticated returns True, the user is authenticated; if the method returns False, you can redirect the user to Microsoft's Passport site for authentication, or offer to sign them into Passport directly from your site.

Security

Optional parameters can be specified to increase the security according to your own site policies. For example, you can ask Passport to verify that the user has signed in using a secure method using encryption or digital certificates, as opposed to single unencrypted text passwords.

Access control list

Once you've verified that the user has been authenticated to Passport, you can query the Passport Manager object to acquire the user's Passport Unique ID, a hex string. That's the only information that you can learn about the Passport user, without asking for additional permissions from the user; however, that PUID code can be used as the basis for your own access control list.

Privacy

If more needs to be known about the user, the Passport Manager's HasProfile can be

	queried to determine if there is a profile associated with that PUID, and then use the Profile method to acquire specific data fields from the Passport server, such as first and last name, e-mail address, birth date, nickname, street address, occupation, gender, time zone, and whether the user has subscribed to Microsoft's Wallet payment service. For more info on how to implement Microsoft passport see (Microsoft, 2004). Several other SSO initiatives exist such as MYUID (MYUID, 2004) and the Liberty Alliance Project (LAP).

5.7 Discussion

There are a number of general issues related to the definition and format of bridging patterns.

Usefulness

The question remains how detailed or elaborated a bridging pattern needs to be. Experienced software engineers may find bridging patterns trivial and dismiss them. For beginners they might prove useful. Showing an implementation part in terms of classes and objects may not communicate well outside the SE community but may be quite valuable for software engineers in understanding the problem better. In addition, there may be problems when describing the implementation part because the implementation may be different for the type of system (for example web based or desktop). For example Cancel in regular applications is often differently implemented than in a web application (where it is often not supported apart from the Cancel button in a browser). The issue of detailedness of a pattern is a general ‘problem’ in design pattern research. Nonetheless, we feel that the level of detailedness we show in our example patterns allows the patterns to be useful for both UI designer and software engineers/architects while not being too detailed.

Architecture sensitivity

The architectural impact of a pattern on given system very much depends on what already has been implemented in the system. If some toolkit or framework is used that already supports a pattern (such as for example struts framework supporting model view controller), the impact may be low while the impact can be high if no pattern support is there. The criterion that we used to define the architectural sensitiveness is whether it has an impact on the software architecture. Architectural sensitivity is a relative notion and depends on the context in which a pattern is being applied. It is therefore the responsibility of the pattern writers to identify the main dependencies together with an explanation of how they influence the architectural sensitivity. A goal of architecture design and analysis is therefore to get rid of the irreversibility (Fowler, 2003) in software designs. To some extent it should be possible to design an architecture that allows one to add usability solutions without much effort during late stage if need be.

Architectural impact

The architectural impact can work on different levels in the architecture. Bosch (Bosch, 2000) identifies four types of architectural transformations e.g. architectural style, architectural pattern, design pattern and transformation of quality requirements to functionality. The impact of a bridging pattern can be either one of those transformations or a combination. Each of such a transformation has a different impact on the architecture. Imposing an architectural style is a transformation with a major architecture wide impact (Bosch, 2000). Multichannelling can be implemented using n-tier architectures which are examples of a layering style (Buschmann et al, 1996). An architectural pattern does not generally reorganize the fundamental components but rather extends and changes their behavior as well adds one or a few components that contains functionality needed by the pattern. Multichannelling can also be facilitated using the architectural pattern MVC (Buschmann et al, 1996). MVC adds view and controller components to the current architecture, which is considered to be the model since it contains the domain functionality. Applying a design pattern (Gamma et al 1995) generally affects only a limited number of components in the architecture; however the impact can still be very high depending on the size of the components. An example of a design pattern implementation is the undo which can be facilitated using the command pattern. The last transformation deals with extending the system with additional functionality not concerned with the applications domain but primarily with improving the quality attributes. The encapsulation strategies for providing SSO are an example of such a strategy. This type of transformation (such as adding a wrapper around existing security functionality) may require minor reorganizations of the existing architecture but may still be expensive to do during late stage design.

Creating bridging patterns

When creating bridging patterns only a subset of ID patterns can be meaningfully converted to bridging patterns. Many ID patterns do not require any significant programming that can generically be described in a bridging pattern, e.g. Mode Cursor or Warning (Welie and Trætteberg, 2000) pattern. The ID patterns that exist are of different levels (ranging from detailed interface issues to large scale interface and interaction issues such as providing consistency) so it should not be expected that all patterns can be re-written as bridging patterns. Even for patterns where it is possible to create a bridging version, the implementation description is bound to be very situational (e.g. depending very much upon which UI toolkits/ libraries are used). It remains the question whether this information is still useful then. To create bridging patterns software engineers and HCI engineers should work together closely.

5.8 Future Work

Our bridging pattern format and the examples we have shown are not intended to be exhaustive. For specific application domains different implementations may be possible. Therefore a number of domains specific implementations may need to be provided. Also the architectural sensitiveness of some patterns for some applications domains is open to dispute. As identified with the Undo pattern often some sort of framework is necessary that uses design- or architectural patterns. Without a sufficiently detailed description of the framework that is needed for the implementation, the architectural description lacks credibility.

Practice shows that a number of ID patterns such as Auto save (Laakso 2004), Cancel (Bass et al, 2001, Workflow patterns), Interaction History(Common ground, 1999) and Scripted Action Sequence/Macros(Common ground, 1999) are often implemented using the same implementation framework that is needed to facilitate Undo. Other frameworks such as the framework needed for Multichannelling also provide ID patterns such as Preview (Welie, 2003), Container Navigation (Welie, 2003) and Overview beside Detail (Laakso 2004). We suspect that some clusters of ID patterns are facilitated by the same architectural framework. This is very interesting since often engineers are not aware multiple ID patterns can be facilitated using the same implementation framework.

In this paper only four patterns are presented. Future work should lead to the expansion and reworking of the set of patterns and validating our assumptions that clusters of patterns exist that share the same implementation framework. Concerning the purpose of bridging patterns e.g. using them for architectural assessment three case studies (Folmer et al, 2004) have been performed (using a pattern framework that consists of more patterns but these patterns are have less detail than the bridging patterns presented in this paper). The experiences (Folmer et al, 2004) with a pattern based approach for architectural assessment were good. In some cases architects changed their software architecture to provide support for these patterns.

This paper has presented a new type of pattern called a bridging pattern. Bridging patterns extend interaction design patterns by adding information on how to generally implement this pattern. For four patterns (selective undo, multi-channel access, wizard and single sign-on) the generic implementation and architectural considerations are presented. Bridging patterns can be used for architectural analysis: when the generic implementation is known, software architects can assess what it means in their context and can decide whether they need to change the software architecture to support these patterns. Such architectural analysis can take place during design phases but also during product evolution when new requirements arise for new releases of the system. In addition bridging patterns may aid communications across the SE and HCI boundaries. By discussing how a bridging pattern can improve usability and what its effect may be on the architecture, the mutual awareness of the restrictions that exist between SE and HCI can be raised during requirements analysis. This may prevent part of the high costs incurred by adaptive maintenance activities once the system has been implemented and leads to architectures with better support for usability.