A Case Study on Adaptability Problems of the Separation of User Interface and Application Semantics

Marc Evers

University of Twente, Dept. of Computer Science, Software Engineering Group
P.O. Box 217, 7500 AE Enschede, The Netherlands

e-mail: evers@cs.utwente.nl

16 August 1999

Abstract

A large number of software architectures for interactive have been described in literature, like the Seeheim, PAC-Amodeus, and Model-View-Controller architectures. Most of these architectures are based on the traditional view of interactive software, namely the view that an interactive software system can be separated in an application part and a user interface part. The application part contains the functionality of the software – what the system does – and the 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, adaptability, portability, complexity handling, and separation of concerns of interactive software.
The principle of separating interactive software in application and user interface parts has its merits. It can however lead to serious adaptability problems in software that provides fast, frequent and intensive feedback, in particular semantic feedback. Semantic feedback refers to feedback the system gives to the user concerning the semantics of the application, i.e. the objects that the user perceives and manipulates. In these systems, the boundary between application and user interface becomes less sharp and the semantics of the interface and the application tend to be highly coupled. Examples of such interactive systems are direct manipulation systems, virtual reality systems, and systems with natural language interfaces.
The problem of semantic feedback is that it is functionality that has both application and user interface aspects and crosses the application-interface boundary. It requires excessive use of semantic information from the application part and in this way, it compromises the separation of application and user interface concerns. As a result, it becomes more difficult to modify or extend functionality related to semantic feedback. In other words, the adaptability of the interactive software is decreased.
In this report, a case study concerning an interactive CD database system is used to illustrate the adaptability problems of separation-based architectures for interactive software.

Table of contents

1 Introduction and problem statement 3
2 Background 3
2.1 Feedback in interactive systems 3
2.2 Adaptability 4
2.3 Models and architectures for interactive systems 5
2.3.1 Linguistic model 5
2.3.2 The Seeheim model 6
2.3.3 PAC-Amodeus 6
2.3.4 Model-View-Controller 7
2.3.5 Higgens UIMS 8
2.3.6 Other architectures 9
3 Case study – CD database system 9
3.1 CD database system 9
3.2 Application of the user interface architectures 10
3.2.1 The functional core 10
3.2.2 PAC-Amodeus architecture 11
3.2.3 Linguistic model 14
3.2.4 Model-View-Controller 15
3.2.5 Higgens UIMS 16
3.3 Extensions and changes to the CD system 17
3.3.1 Introduction of distribution and concurrency 17
3.3.2 Adding a picture of the artist 18
3.3.3 Adding exception handling 19
3.3.4 Changing the sorting algorithm 20
3.3.5 Changing the type of progress indication 20
3.4 Implementation issues 20
4 Evaluation 21
5 References 23

1 Introduction and problem statement

The role of human computer interaction has become quite important for software. Human end users play an active and essential role in the operation of interactive software and user interfaces have become an essential part of many software systems.
Myers and Rosson describe the results of a survey of user interface programming. These results show that in current applications, almost half the code is devoted to the user interface part of software and during design and implementation about half of the time is spent on the user interface; during maintenance, about one third of the time is spent on the user interface [Myers, Rosson, 1992].
To cope with the software engineering aspects of interactive software, several software architectures for interactive software have been developed. These architectures are also called user interface architectures or user interface management systems (UIMS). The objective of user interface architectures is to make the development and maintenance of interactive software more effective and efficient by providing separation of specific user interface related concerns.
Most user interface architectures are based on the assumption that the functionality and the user interface are separate, distinguishable concerns that should be separated [Edmonds, 1992]. The functionality is what the software actually does and what information it processes, and is also called the *functional core*. The user interface defines how this functionality is represented to human end users and how user input is processed.
These user interface architectures have been proven useful, but they also introduce problems with respect to the adaptability of the interactive software. These problems occur in particular in 'highly' interactive systems, where user interface and application are semantically dependent on each other: the more interactive a software system is, the more coupling there is between its functionality and its user interface. Examples of these highly interactive systems are systems that provide their users with frequent and extensive feedback, such as direct manipulation systems and virtual reality environments.

This report describes a small case study to illustrate some of the problems in software engineering for user interfaces mentioned above. I will focus on a particular, important area of software engineering, namely the adaptability of software. The concept of *semantic feedback* will be used to show that the separation of user interface and functionality can cause adaptability problems. Examples of semantic feedback are validation and input masks: when a user provides input, the input is validated immediately. First, background work will be described section 2. Then I will discuss a CD database system as an example application in section 3. An evaluation is given in section 4.

2 Background

In this section, I will describe background work that is relevant for the case study. First, I will describe an important issue in interactive systems that complicates the software engineering aspects of user interface design and maintenance, namely feedback. Then I will give a short survey of the concepts of adaptability and change of software. Last, I will describe a selection of the several different architectures and models for the design and implementation of interactive systems that have been introduced in the last fifteen years.

2.1 Feedback in interactive systems

The problems that I address in this case study occur in particular in interactive software systems that are characterised by frequent and intensive feedback, especially semantic feedback. Examples are direct manipulation systems, virtual reality systems, and systems that use natural language techniques (see e.g. [Nielsen, 1993]). Specific examples of semantic feedback are error checking and recovery, validation, and generation of default values ([Dance et al., 1992]; [Cunningham, 1995]).
Feedback is often a pervasive and important concept in these interactive systems and plays an essential role for the usability of user interfaces. The importance of feedback is stressed, amongst others, by Foley et al. *Semantic feedback* refers to feedback the system gives to the user concerning the semantics of the application, i.e. the objects that the user perceives and manipulates [Foley et al., 1990]. High levels of semantic feedback require excessive use of semantic information on the functional core.
In direct manipulation interfaces, feedback is essential as well. In [Görner, Vossen, Ziegler, 1992], direct manipulative systems are characterised by a high degree of interactivity, although a direct manipulation interface can also have components with a low level of interactivity. Such systems provide immediate interpretation of the actions of the user, as well as direct and continuous feedback, both during input and processing. Changes made are reflected immediately.

Versendaal has identified a number of complications that can occur when the user interface and the application semantics are separated [Versendaal, 1991]. He describes the sequencing of tasks of an interactive system as follows: at an abstract level, it can be seen as a user input task, followed by a computer task, followed by output tasks. The input and output tasks are performed by the user interface component, the computer tasks by the application component. The following three complications have been identified:

- presentation of semantic feedback during the input task; an example is constraints on the input that are part of the application semantics;
- execution of an output task during the computer task execution (in other words, intermediate output); an example is the status of a computation or progress indication;
- execution of a computer task that only deals with the user interface and not with the application semantics; an example is a complex graphical system where specific computer tasks manipulate the objects that are shown on the screen.

These complications are related to feedback. Note that direct manipulation interfaces can be seen as an instance of the third complication.

To cope with semantic feedback, there are two main classes of solutions in a software system based on the separation of functionality and user interface. First, the concern of semantic feedback can be put in the user interface part. This approach is called *semantic delegation* or *domain knowledge delegation* [Coutaz, Nigay, Salber, 1995]. Second, the semantic feedback concern can be put in the application part, resulting in a lot of communication between the interface and the application.
The separation of user interface and application has also been subject of many discussions, see e.g. the report on the CHI'88 panel discussion on User Interface Management Systems [Rosenberg et al., 1988]. One view expressed in this discussion is that, in the case of direct manipulation systems, the strict separation will lead to duplication of parts of the applications, or elimination of the separation through ad-hoc programming.
Hartson describes the above as a trade-off between the amount of separation (related to the *cohesion* of the different components of the software system) and the amount of communication (*coupling*) between functionality and user interface [Hartson, 1989]. According to Wiecha et al., semantic feedback makes an interface depend on extensive knowledge of the application, making strict separation hard [Wiecha et al., 1989].

2.2 Adaptability

The concept of adaptability is quite important in the areas of software engineering and software maintenance. Adaptability is usually defined as robustness to change. Related and overlapping concepts are extensibility, maintainability, and modifiability.
The relevance of adaptability and making software adaptable becomes clear: about 70% of the effort for building software is spent during the maintenance phase, thus on making changes related to errors, changed requirements, and enhancements of the software (see e.g. [Pressman, 1992] and [Fenton, Pfleeger, 1996]). Furthermore, about one third of the time of the maintenance phase is spent on the user interface [Myers, Rosson, 1992].
It makes no sense to talk about adaptability of software in general terms: a software system is adaptable (or not) in specific ways to specific changes and modifications ([Fayad, Cline, 1996]; [Bass, Clements, Kazman, 1998]). A more precise and operational definition of adaptability of a software system (or a part of it) would be: the anticipation of specific changes and variations of the system, where anticipations of a specific change means building the software in such a way that performing the change will have minimal impact on the system and will be of minimal cost.
The process of performing changes to a software system can suffer from two adaptability problems: *scattering of a change* and *propagation of a change*. Scattering of a change means that a change to a single concept of the system requires modification of several components. Propagation of changes means that a modifying a single component requires modifying several other components as well, because of direct and indirect dependencies[1]. These two problems are related to the concepts of *focus* and *scope* of changes to software, as described by Ecklund et al. [Ecklund et al., 1996]. The focus of a change is defined as "the set of system responsibilities that the change directly affects", while the scope of a change "refers to the pervasiveness of ramifications of the change throughout the artifacts developed". A large focus of a change means that it is scattered; a large scope means that the change suffers from propagation throughout the system.

[1] Concrete examples of components and dependencies are classes and associations, procedures and procedure calls.

In terms of these adaptability problems, providing adaptability for a specific change means building the software in such a way that performing this change will not lead to scattering and propagation of the change. Several techniques and mechanisms are available to make software adaptable. Examples are modularity, encapsulation, inheritance, and design patterns.
Note that unanticipated changes cannot be anticipated. Whether a system is adaptable with respect to certain unanticipated changes can only be determined after the fact.

Adaptability and user interfaces

In the context of user interfaces, there are three main sources of change and variation: changes in the user population of the system, changes in the context of the software system, and changes in the functionality. I will elaborate on these three sources.
The population of a software system can be heterogeneous and dynamic. It can be heterogeneous in the sense that there are different users, users differ in age, personality, working style. Different types of users may perform different sets of tasks. Cultural diversity among users is a different source of heterogeneity. According to Reynolds, it is hard to design a good interface for a specific cultural group, and it is better to provide some means for individualisation, customisation, and adaptation of the interface [Reynolds, 1997].
The user population can be dynamic in the sense that users learn when using the software system. As a result their usage patterns change and they may need a different interface. Furthermore, in a distributed environment, users are not confined to a single workspace and can be mobile.
Not only the user population of a software system, but also the environment or context in which the system operates can be heterogeneous and dynamic. An example of a heterogeneous context is the availability of different media for the end systems: the end system can be a workstation, a simple text terminal, a web-based terminal, or even a voice-response system. An example is the TAST-system recently introduced at the University of Twente. TAST is a system with which students can get information about exams and register themselves for these exams. It uses two different media for interfacing with users: a web-based interface and a voice-response interface through telephone[2].
An example of a dynamic context is the varying quality of service provided by the underlying network.
The functionality of software usually evolves as a result of changing requirements. This is one of the large problems in the field of software development. Changes in the functionality will usually affect the user interface. One would like to keep the required rework on the user interface minimal.

2.3 Models and architectures for interactive systems

In this section, I will describe the following user interface architectures: linguistic model, Seeheim model, PAC-Amodeus architecture, Model-View-Controller, and the Higgens UIMS. I will conclude this section with some remarks on some other user interface architectures.
All these architectures are based on the assumption that the user interface and the application semantics can and should be separated. Note that the different architectures described here are quite similar. This also holds for many other user interface architectures that are not presented here. Many architectures can be seen as different variations on the same theme, namely the Seeheim architecture ([Edmonds, 1992]; [Bass, Clements, Kazman, 1998]).
A thorough analysis of different user interface architectures and the rationale behind them is presented in chapter 4 “Software Architecture Models” of [Gram, Cockton, 1996].

2.3.1 Linguistic model

In the linguistic architecture of interactive software, the following layers are generally distinguished: lexical, syntactic, and semantic layers [Foley et al., 1990]. Note that I apply the linguistic concepts to the *software engineering* aspects of the user interface (the ‘inside’ view). The linguistic concepts have also been applied to the human-computer interaction aspects of the interface (the ‘outside’ view).
The semantic layer describes the object that the user perceives and manipulates. The syntactic layer describes the syntactic elements with which the objects and actions of the semantic layer are expressed. The lexical layer defines the tokens of interaction. The three linguistic layers are added to the functional core. The figure below illustrates the linguistic architecture.

[2] TAST stands for ‘Tentamen Aanmeld Systeem Twente’; the URL of the web-based interface is http://www.utwente.nl/tast (in Dutch)

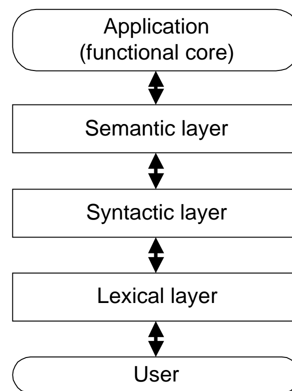

Figure 1. Linguistic architecture of interactive software

It is possible to generalise the architecture described above. An example of such a generalisation is the architecture presented in [Hoffner, Dobson, Iggulden, 1990]. In this architecture, the user interface consists of a number of *languages* (layers) that try to bridge the gap between the 'language' of the user and the 'language' of the application. The number of languages depends on how large this gap is. Another generalisation is to apply the semiotic concepts of *expression* and *content* (see e.g. [Andersen, 1997]). These concepts can be applied in a hierarchical way: the content at one level is in fact the expression of another level.

2.3.2 The Seeheim model

The Seeheim model can be seen as the basic UIMS model. It distinguishes three components of the user interface: the application interface, the dialogue controller, and the presentation layer. The application interface is the view that the user interface has of the application semantics, it can be seen as an abstraction of the actual implementation of the tasks. The dialogue controller defines the structure of the dialogue; the representation component contains the actual representation of the interface ([Green, 1985]; [Versendaal, 1991], [Treu, 1994]). The figure below depicts the Seeheim model.

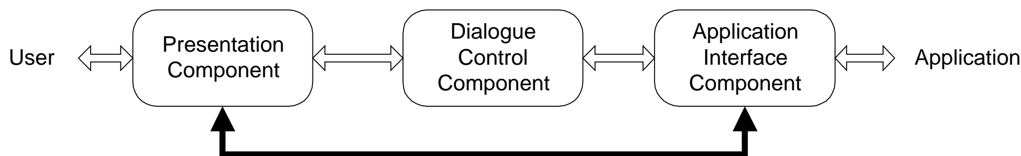

Figure 2. Seeheim model

The Seeheim model was first introduced in [Green, 1985]. In this paper, the three components are described as follows. The presentation component takes care of the external representation of the user interface. It handles e.g. physical input (and output) devices and it provides a kind of mapping between the external (physical) representation and the internal, abstract representation of input and output.
In the dialogue control component, the structure of the user-computer dialogue is defined. In contrast to the presentation component, which is stateless, the dialogue control has a state. The application interface component represents the application from the viewpoint of the user interface. It reflects the semantics of the application. Note that the need has been identified for the presentation and application interface components to be able to bypass the dialogue control component and communicate directly.
Techniques for representing the dialogue control component are e.g. state transition diagrams, grammar-based techniques, and event-based models. Note that not only are representations needed for the three components, but for the interrelations and interactions between them as well.
The layers of the Seeheim model reflect linguistic levels: the presentation component defines lexical aspects, the dialog control component defines syntactic aspects, and the application interface component defines semantic aspects.

2.3.3 PAC-Amodeus

The PAC-Amodeus model is a combination of two models, the Arch model and the Presentation-Abstraction-Control (PAC) model. The Arch model, which is a refinement of the Seeheim model, distinguishes five components of an interactive software system ([Encarnação, 1997]; [Coutaz, Nigay, Salber, 1995]; [Coutaz, 1997]). The structure of the Arch model is shown in the figure below.

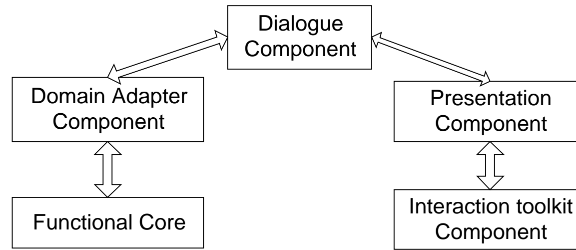

Figure 3. Arch model of interactive software

The *functional core* (or domain-specific component) encapsulates the functionality of the system and offers an abstraction of the application semantics. The *interaction toolkit component* encapsulates the physical interaction. It includes e.g. platform-dependent implementation of user interface widgets as well as hardware aspects. The *presentation component* provides an abstraction from the interaction toolkit component and provides platform-independence. The *dialogue component* contains the task level sequencing and provides the mapping between domain specific objects and user interface specific objects. The *domain adapter component* adds to the functional core any domain related behaviour that is needed by the dialogue component.

The dialogue component, domain adapter component, and the domain specific component communicate through *domain objects*. The dialogue component and the presentation component exchange *presentation objects* (abstract, virtual objects representing aspects of interaction). The presentation component and the interaction toolkit component exchange *interaction objects*.

Note that the domain adapter and presentation components can be seen as instances of the Adapter design pattern as described in [Gamma et al., 1995]. The intention of the Adapter design pattern is to convert the interface of a class or a subsystem into another interface that is expected by a client.

In the PAC-Amodeus architecture, the dialogue component is refined using the Presentation-Abstraction-Control (PAC) model ([Coutaz, 1987]; [Coutaz, 1990]; [Coutaz, Nigay, Salber, 1995]; [Coutaz, 1997]; [Buschmann et al., 1996]). PAC is an multi agent model that structures the dialogue component as a *hierarchy of interacting agents*. An agent corresponds to e.g. a window, a group of widgets, or an individual widget. Each agent has three facets: an *abstraction* facet, which contains the data or objects; a *presentation* fact, which encapsulates the presentation logic of the agent; a *control* facet, which controls the communication between abstraction and presentation and the communication between subordinate and superordinate agents. An example is shown in the figure below: a top-level agent represents the application as a whole, while its subordinate agents represent different dialogue windows.

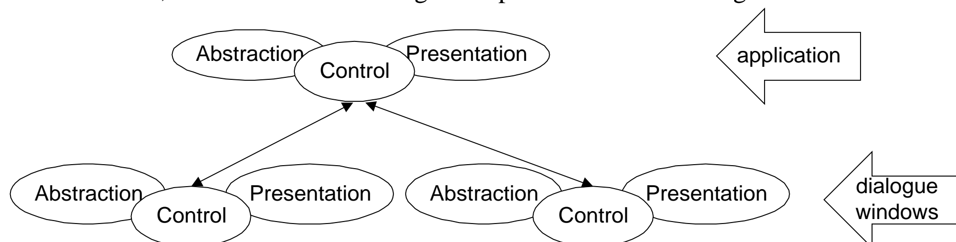

Figure 4. Example of a PAC hierarchy

2.3.4 Model-View-Controller

The *model-view-controller* (MVC) architecture is a well-known model that has been applied for instance in the Smalltalk environment [Krasner, Pope, 1988]. It 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; the controller knows how the user interactions with the view will affect the information in the model. If the model changes, it notifies its views of this change. This is an application of the Observer design pattern described in [Gamma et al., 1995]. The model itself is linked to the functional core. The view and controller are tightly coupled and are often combined into a single object. Note that there is a lot of coupling and interdependency in instances of the MVC architecture. The MVC architecture is depicted in the figure below.

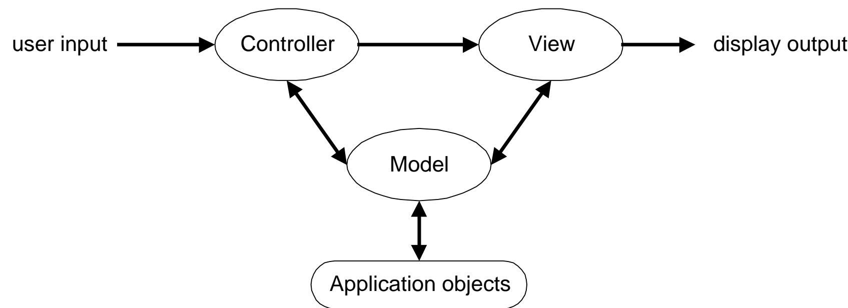

Figure 5. MVC model

In [Buschmann et al., 1996], the MVC architecture is described as an architectural pattern for interactive systems. The MVC pattern separates the functional core from the interface; furthermore, in the interface, the input and output is separated using controller and view objects respectively. In this way, it tries to resolve the following forces:

- the need for multiple, synchronised views on the same information;
- the need for adaptability of the user interface, even at runtime;
- a different 'look and feel' or porting the interface should not affect the functional core of the application;
- data manipulations should be reflected immediately by the display and behaviour of the application.

Disadvantages ('liabilities' in Buschmann's terminology) of MVC are:

- MVC increases the complexity of the system;
- view and controller tend to be intimately connected;
- close coupling of views and controllers to the model;
- the change propagation mechanism used can result in an excessive number of updates;
- data access in views can be inefficient;
- views and controllers will need to be changed when the system is ported;
- MVC is often difficult to use in combination with user interface tools.

2.3.5 Higgens UIMS

The Higgens UIMS, described in [Hudson, King, 1988], is a user interface architecture which has explicitly been designed to support, among other things, semantic feedback. The Higgens system is actually a system that generates a user interface according to specifications. We will not focus on these specifications or the generation process, but on the structure of the generated user interface and the ideas behind it. The global structure of the Higgens generated user interface is show in the figure below.

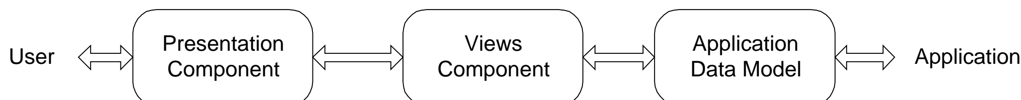

Figure 6. Global structure of Higgens generated user interfaces

The emphasis in this structure is on the objects of the application and their representations to the user. These representations are represented by the views component. The presentation component manages the actual input and output devices. The application data model provides an interface to the application itself. To deal with semantic feedback, an explicit model of the application semantics is generated. This application semantics model consists of entities with attributes and relations between the entities. This model can also contain attribute evaluation rules and constraints (or assertions) on the attribute values. Note that this approach of providing an explicit model of the application data and using this model for semantic feedback can be seen as an instance of structural reflection on the application objects.

A view of (a part of) the application is constructed by graph traversal of the application. For each node in the application graph a corresponding view node can be constructed. In this way, a separate viewing tree is constructed that is linked to the application data. This viewing tree can be updated incrementally. To create and maintain the viewing tree, not only the application data itself but also the meta-data can be used. The viewing tree communicates with the presentation component of the system. The viewing tree is causally linked to the application: if the user changes something, this is reflected to the application objects.

The figure below shows an abstract example of an application objects graph, a viewing tree and some presentation objects.

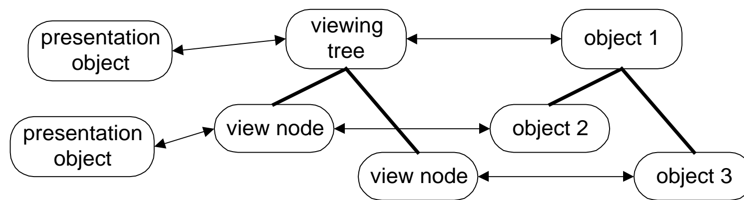

Figure 7. Example Higgens user interface structure

2.3.6 Other architectures

The Model-View-Controller architecture is used in commercial software development environments, like VisualWorks Smalltalk. In other development and programming environments the separation between application and user interface can often be found. The Microsoft™ Foundation Classes for instance provide the *Document-View architecture*, where the document represents the application data and contains the knowledge on persistent storage; the view knows how to render the data from the document and how to handle interaction with users. A document can have multiple views [Microsoft Developer Network].

The *Swing* library of the Java 1.2 language offers separation of the *application*, a *model* of the relevant information from the application, the *view*, and the specific *look-and-feel* of the user interface. The user interface components of the Swing library communicate with models that should have specific interfaces. These models are usually abstractions from specific application information. The user interface components themselves encapsulate the specific look-and-feel ([JDK 1.2], Java Foundation Classes (JFC) – Swing Components).

3 Case study – CD database system

In this section, I will describe the subject of this case study, a compact-disk database system. I will focus on two particular parts of the system, finding a CD and sorting the CDs. First, I will describe the system (section 3.1) followed by a description of the results of applying the user interface architectures to the case (section 3.2). After having presented the basic system, I will propose a number of possible extensions and changes to the system and I will analyse how these extensions and changes affect the CD system and how the user interface architectures handle these changes (section 3.3). I will also look briefly at the issue of implementing user interface architectures in section 3.4.

Note that I will not work out the Seeheim model, as it is more or less contained in the PAC-Amodeus architecture.

3.1 CD database system

The subject of the case study is a simple CD database system that needs to be designed and built. The system manages a list of CDs, keeping information on the artist and the title of each CD. The user can add CDs to the list, remove CDs, look at the list of CDs, and sort the list. Furthermore, the user can search for a specific CD. The CD database system has a graphical user interface (a WIMP[3]-interface). There is a main window from which all the functionality is available. I will not describe the main window, but I will focus on two aspects of the system that are available from the main window: finding a CD and sorting the list of CDs.

A separate dialogue window has been defined for finding a CD. The user types the artist name or CD title in edit boxes. The system tries to assist the user in the following way: when the user types part of the artist name, the system shows a list of *suggestions* from which the user can select one. The list of suggestions consists of the artist names that begin with the letters typed by the user. The ‘Find CD’ dialogue window has two edit fields, a selection list, an ‘OK’ button, and a ‘Cancel’ button, as shown in the figure below.

[3] WIMP stands for ‘Windows, Icons, Menus, Pointer’ and refers to common graphical interfaces that use these elements.

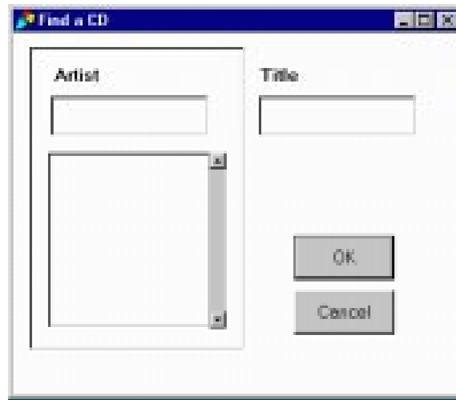

Figure 8. 'Find CD' dialogue window

During the sorting of the list, a *progress indicator* is shown. The progress indicator displays a percentage that provides the user with an indication of the progress of the sorting process. A progress indicator is especially useful when an algorithm or a process can take a lot of time to execute. Progress indicators show the user that the system has accepted the user's request, and that it is busy executing the request. Progress indicators give experienced users the opportunity to plan their time better. Myers presents some experimental evidence for this in [Myers, 1985].
The sorting dialogue window shows a bar and a percentage as progress indication, as an indication of e.g. how many items have been sorted. Furthermore, there is a 'Cancel' button for interrupting the sorting process. The dialogue window is shown in the figure below.

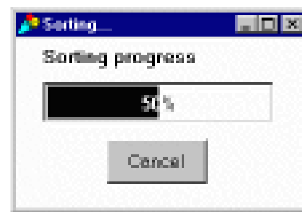

Figure 9. Sorting dialogue window

In Versendaal's terminology, the suggestion list is an instance of semantic feedback during user input and the progress indication is an instance of output during processing [Versendaal, 1991].
One of the concerns in constructing the user interface for finding a CD is creating a sublist of artist names when the user has typed something. This concern has both application and user interface aspects. A second concern is the actual finding of one or more CDs that satisfy the criteria entered by the user.
The main concern in the user interface for sorting the list of CDs is to get information on the progress of the sorting process. This information has to be extracted from the execution of the sorting algorithm; a kind of 'hook' is needed for this purpose. Note that progress is a property of the sorting algorithm that emerges in the context of the sorting algorithm and the user.

3.2 Application of the user interface architectures

In this section, I will present design models of the CD system that result from applying the PAC-Amodeus, the Linguistic and the Model-View-Controller architectures to the case. I am less interested in the lower level user interface issues and therefore I assume that a kind of presentation interface exists that offers 'abstract' user interface components (widgets), like labels, buttons, edit boxes, and selection lists. This presentation interface abstracts from physical devices, the specific windowing system and the 'look and feel'.

3.2.1 The functional core

The three architectures that I apply all assume that there is a part of the system that contains the functionality that is more or less independent of the user interface. In this section I will present an object-oriented model of this functional core of the CD database system.
The functional core of the CD system consists of a CD collection that contains zero or more CD objects. A CD object has two attributes, the name of the artist and the title of the CD. The CD collection has methods to add and remove CDs, and a set of methods to iterate the collection (e.g. goFirst, goNext,

current, isLast, and isEmpty). It also has a sort method to sort the list of CDs. The class diagram of the functional core is shown in the figure below.

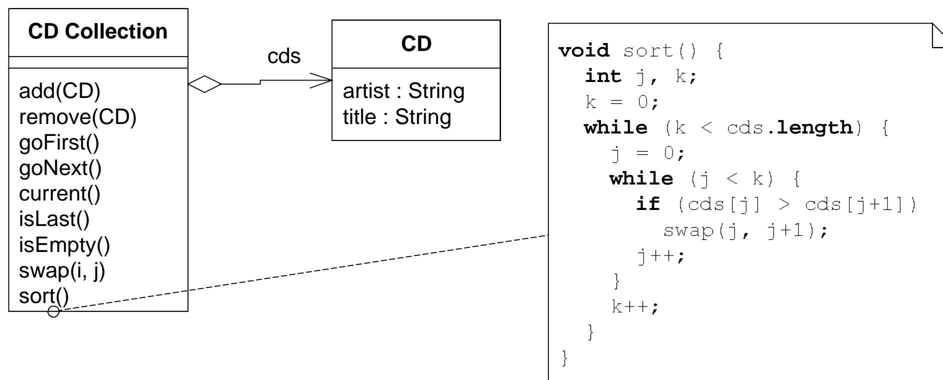

Figure 10. Class model of the functional core of the CD system

On progress indication for the sorting algorithm

In order to give the user an indication of the progress of execution of the sorting algorithm, information about the progress has to be extracted from the algorithm. Progress is not really an inherent property of the sorting algorithm, but it emerges in the context of the sorting dialogue window. The way in which the progress is computed depends both on the internals of the sorting algorithm and on the kind of progress that needs to be displayed, e.g. time or number of elements processed.

In the class diagram above, a Java implementation of the bubble sort algorithm is shown as an example. To determine the progress in this algorithm, the value of loop counter *k*, the number of elements to be sorted, and some information on how these values are used in the algorithm, are needed to determine the progress percentage. The outer loop is executed *cds.length* times (the number of elements to be sorted). A progress percentage can be computed from *k* and *cds.length* as follows: *(1 – (k / cds.length)) * 100%.*

For the extraction of runtime information it is possible to use e.g. the Observer design pattern [Gamma et al., 1995]. The algorithm only needs to notify the progress indication of changes in the values; it only needs to know that some object is observing specific components. The impact of the progress indication on the sorting algorithm is not large, although the algorithm needs to make public some of its internals. In other words, encapsulation is broken. The structural part of progress determination forms a worse problem, because the computation is highly dependent on the internal structure of the algorithm.

It is possible to see the extraction of the information from the sorting algorithm as *structural* and *computational reflection* [Maes, 1987]: the progress indicator reasons about the structure of the sorting algorithm and about the computation that takes place. The use of reflection techniques (e.g. associating a meta-object with the sorting algorithm) can remove dependencies of the sorting algorithm on the progress indication. No ‘hooks’ have to be added to the algorithm. Note that the progress indication still remains highly dependent on the specific algorithm.

3.2.2 PAC-Amodeus architecture

I assume that the interaction toolkit component and the presentation component are provided and that the presentation component offers a set of (abstract) widgets and other user interface components. I will focus on the dialogue component, the functional core and the interface between them (the domain adapter). For the dialogue component, I will create a hierarchy of PAC agents.

I propose the following PAC model of the dialogue component: there is a top-level agent for the whole CD system, which will not be worked out. It has a subordinate agent for finding a CD and one for sorting the CDs. It can also have other sub-agents for other dialogue windows.

The agent for finding a CD handles the ‘Ok’ and ‘Cancel’ buttons (see Figure 8), and it has two sub-agents for the artist input and the title input. The ‘title input’ agent encapsulates the label and edit-box for the title. Its abstraction part contains the title entered by the user. The ‘artist input’ agent has two subagents, one for the edit-box and one for the list with suggestions. It co-ordinates these two agents, e.g. if the user selects a suggestion in the list, the artist name is put in the edit box. Its abstraction contains the artist name as entered by the user.

The agent for sorting CDs has a single sub-agent for the progress indicator bar itself.

The hierarchy of agents is shown in the figure below. The greyed presentation facets represent widgets and collections of widgets that can be mapped directly on existing interaction components.

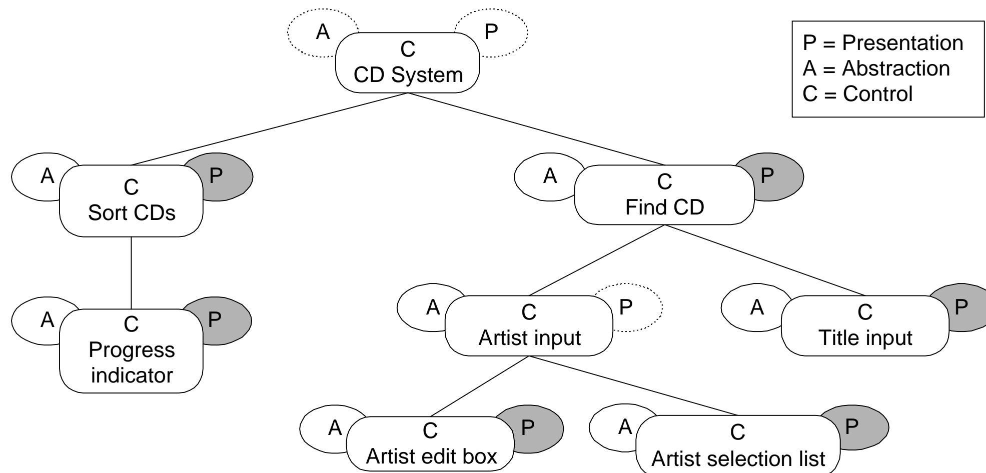

Figure 11. PAC model of the dialogue component of the Find CD dialogue

Note that the 'artist input' agent can be kept generic if one abstracts from the details of the artist. It then becomes a user interface component that lets the user enter a string and that gives a set of suggestions of the user input in a selection list. The agent needs a protocol for retrieving the list.
The table below gives a description of the different agents.

Agent	*Facet*	*Description*
Find CD	Abstraction	contains the artist name and the title of the CD
	Presentation	consists of several interface elements: the dialog window, labels, the 'Ok' and 'Cancel' buttons
	Control	handles the 'Ok' button by showing the selected CD; handles the 'Cancel' button; creates and destroys the dialog window
Artist input	Abstraction	contains the name of the artist that has been entered
	Presentation	this agent has no presentation
	Control	co-ordinates the two subagents: if the input in the artist edit box changes, the selection list needs to be updated, and the other way around
Artist edit box	Abstraction	contains the (partial) name of the artist entered by the user
	Presentation	consists of edit box and label widgets
	Control	links the abstraction and presentation facets
Artist selection list	Abstraction	contains the list of suggestions and the item in the list that has been selected
	Presentation	consists of a selection list widget
	Control	links the abstraction and presentation facets
Sort CDs	Abstraction	this agent has no particular abstraction
	Presentation	consists of the dialog window, the different labels, and the 'Cancel' button
	Control	handles the 'Cancel' button and
Progress Indicator	Abstraction	contains the progress percentage
	Presentation	consists of widgets needed to display the progress bar
	Control	lets the presentation facet update itself when the actual progress changes

Creating the suggestion list

I have identified a number of alternatives for the concern of creating the suggestion list. It can be placed in the functional core, in the domain adapter component, or in the dialogue component. If it is placed in the dialogue component, it can be assigned to different agents: the 'artist selection list' agent, the 'artist input' agent, or the 'Find CD' agent. I will describe these alternatives in more detail.

1) In the *functional core*:
   a) the CD collection is extended with a method that creates a list of suggestions given a prefix;
   b) the functional core is extended with a 'suggestion list' object that knows how to create a list of suggestions by interacting with the CD collection;

2) In the *domain adapter component*: a 'suggestion list' object is created that knows how to create a list of suggestions by interacting with the functional core;
3) In the *dialogue component*:
   a) the *'artist selection list' agent* retrieves the list of names of all artists from the functional core, and selects those names that match the user's input; the controller of the 'artist input' agent triggers this process and provides a prefix of the artist name;
   b) the *'artist input' agent* retrieves the list of artist names from the functional core, and selects those names that match the user's input; this agent gets the prefix of the artist name from the 'artist edit box' and provides the 'artist selection list' agent with a list of suggestions;
   c) the *'find CD' agent* gets a request from the 'artist input' agent, it retrieves the names of all artists from the functional core, creates the list of suggestions, and returns this list to the 'artist input' agent.

Alternative 1.a is less preferable because it adds very specific behaviour to the CD collection, decreasing the adaptability and reusability of the functional core objects.
Alternative 1.b and 2 are similar. The problem here is determining whether the concern of creating the suggestion list belongs to the functional core or whether this concern is a kind of add-on to the functional core.
Alternatives 3.a and 3.c are less preferable; in my opinion, the concern of creating the suggestion list seems to fit best at the level of the 'artist input' agent.
Object models of the alternatives 1.b, 2, and 3.b are depicted in the figures below. Note that not all the presentation components are shown. The agent objects are shown as rounded boxes. Square boxes represent other objects or groups of objects. Edges represent dependencies between objects.

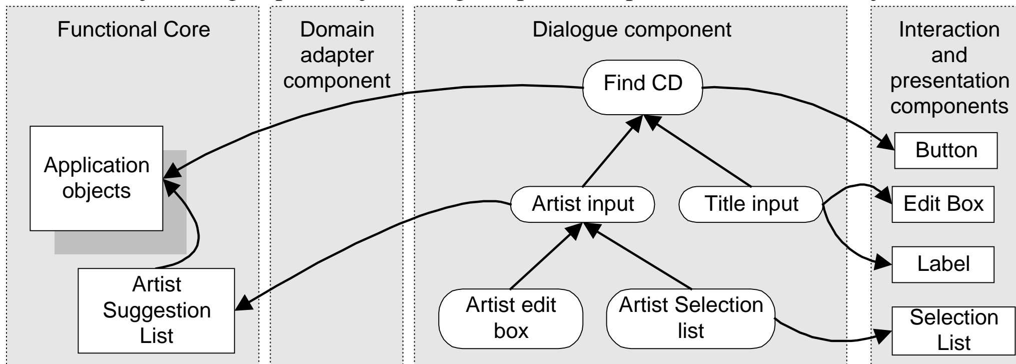

Figure 12. Alternative 1.b

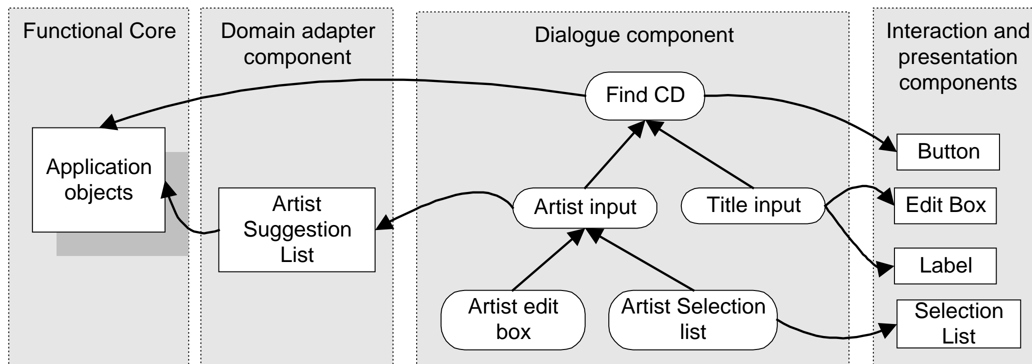

Figure 13. Alternative 2

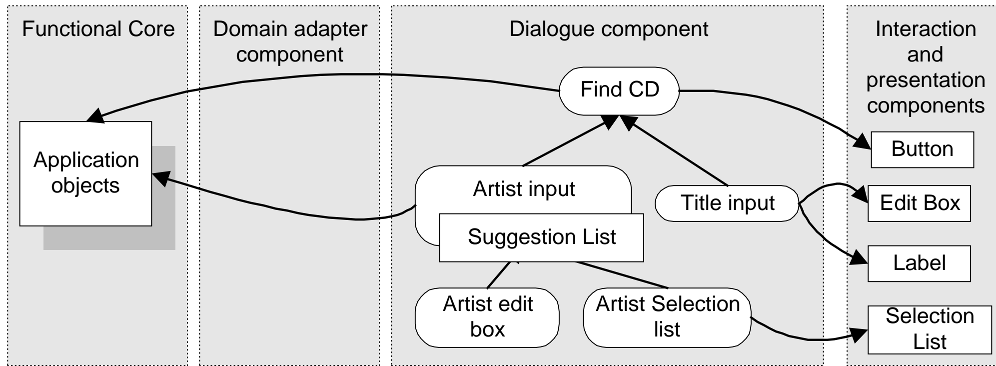

Figure 14. Alternative 3.b

Sorting Progress Indication

The percentage for the progress indicator agent is derived from a 'progress computation' object. This object offers a single percentage or two numbers (the number of completed items and the total number of items). It computes these numbers using a special meta-object that is associated with the sort algorithm. This meta-object monitors the execution of the algorithm and extracts information like the loop counter value mentioned before.
The progress computation object computes the percentages using an internal model of the sorting algorithm (in other words, there is a structural dependency of the progress computation on the sorting algorithm). This is illustrated in the figure below. The agents are shown as rounded boxes. Edges represent dependencies.

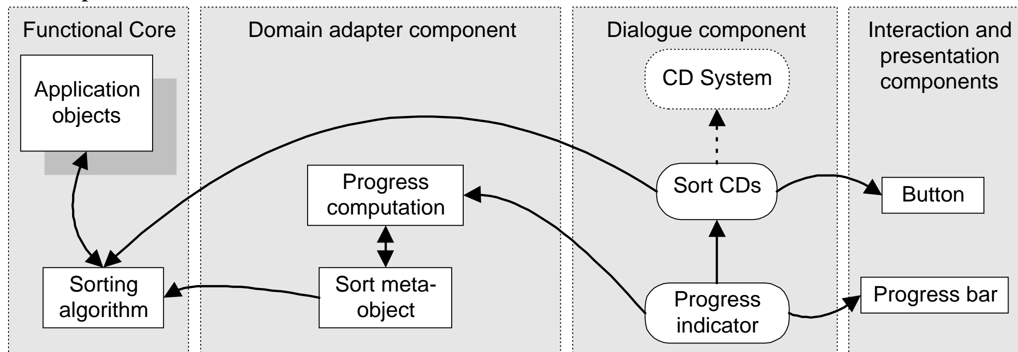

Figure 15. PAC-Amodeus model of the Sorting dialogue

Note that I have placed the sorter meta-object and the progress computation in the Domain adapter component. These objects do not really belong to the functional core, because their purpose is to provide the user interface with appropriate information.
If no reflection techniques are available, meta-objects are not available. Instead, the sorting algorithm has to be extended so that the needed information is made available at the right time to the progress computation. For instance, the line `myProgressIndicator.progress( k )` can be added in the outer while-loop of the bubble sort algorithm described above, so that the value of the loop counter is passed to the progress indicator during each iteration of the loop.

3.2.3 Linguistic model

Following a linguistic approach to model the Find CD dialogue window, I separate the user interface concerns in lexical, syntactic, and semantic layers.
The lexical layer contains the different physical and logical devices and provides an abstraction of the different widgets and devices. It hides the details of the windowing system and offers objects like buttons, edit boxes, and labels.
The syntactic layer defines the components of the Find CD dialogue window: the artist input, the artist suggestion list, the title input, and the 'OK' and 'Cancel' buttons. The 'Find CD Window' object encapsulates the sequencing and management of the dialogue.
The semantic layer defines the link between the application objects and the corresponding user interface widgets. It also defines the semantic feedback (the maintenance of the suggestion list).
A model of the Find CD dialogue window is shown in the figure below. The boxes represent objects, the edges represent dependencies.

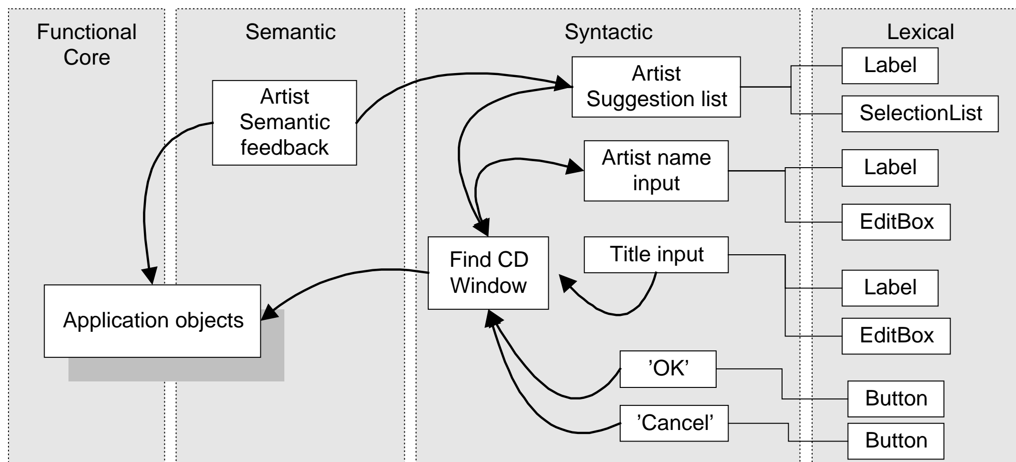

Figure 16. Linguistic model of the Find CD dialogue window

Note that on the one hand, CD objects and the collection of CDs are part of the functional core. On the other hand, these objects are also part of the user's model of the system. In the terms of Foley et al., these objects would be part of the conceptual model [Foley et al., 1990]. The difference between functional core and semantic layer becomes important as soon as there exist multiple views on the application objects. The application objects themselves then belong to the functional core and the views belong to the semantic layer.
The figure below shows the sorting dialogue window according to the linguistic. Like in the PAC-Amodeus case, there is a need for a sort meta-object that observes the sorting process and extracts relevant information, and an object that encapsulates the actual computation of the progress.

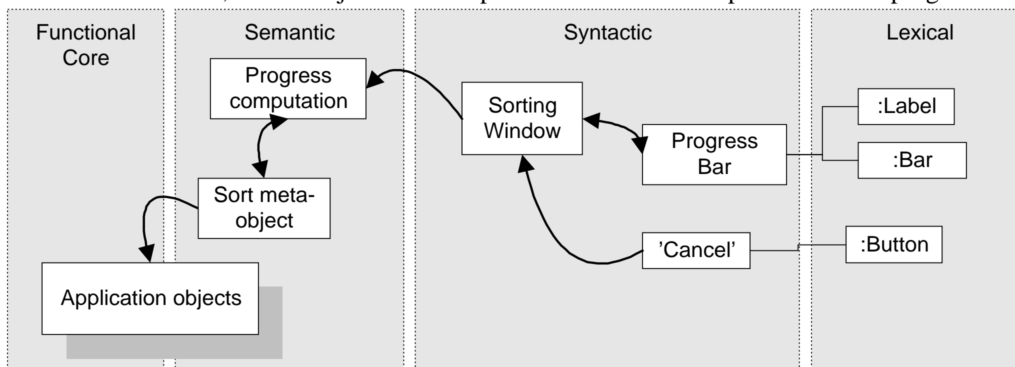

Figure 17. Linguistic model of the Sorting progress window

The notions of semantic, syntactic, and lexical issues are relative to some extent. It is possible to shift the boundaries between the layers. I do not expect however that this will lead to a different set of objects.

3.2.4 Model-View-Controller

The model-view-controller architecture offers a way of composition of MVC triads. As I have mentioned, a view can have one or more subviews. Each subview has its own controller. These subview/controller pairs can be linked to the model or to a specific part of the model. The latter can be called a submodel. The models and submodels perform coordination of the different views and controllers, as well as the linkage to the relevant application objects.
Using this composition mechanism, I propose the following design:

- The 'Find CD' dialogue is represented by a MVC triad.
- The 'Find CD' model has submodels for e.g. the artist suggestion list and the artist input field. The artist suggestion list submodel encapsulates the actual creation of the list based on a prefix typed by the user.
- The 'Find CD' view has subviews for the artist suggestion list and the artist input field.

The (partial) design of the Find CD dialogue according to the MVC model is shown in the figure below. The edges represent dependencies. The dotted edges represent submodel and subview relations. This figure only shows the MVC triads for the dialogue window, the artist suggestion list and the artist input edit box. For readability, I have left out the other components.

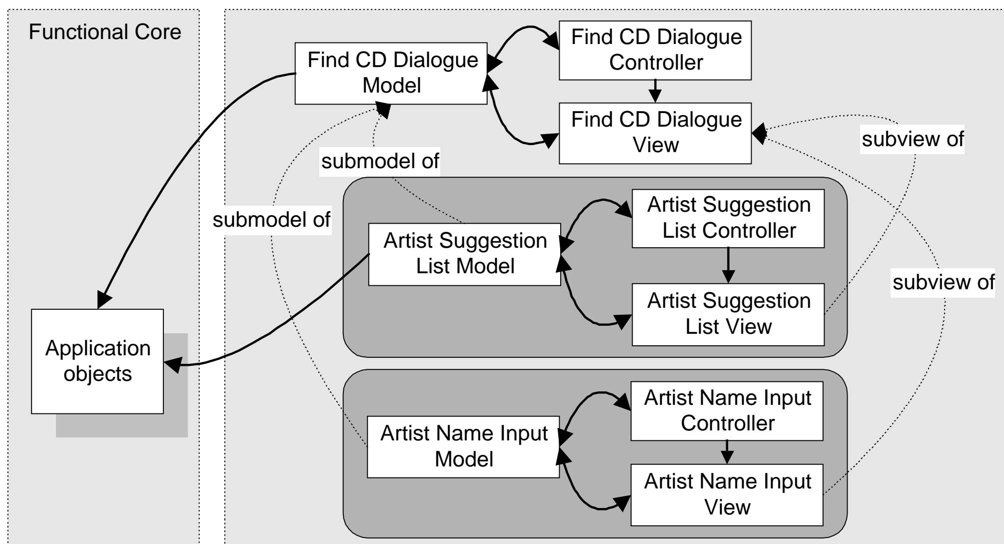

Figure 18. MVC model of the Find CD dialogue window

Applying the model-view-controller architecture to the sorting progress indication will result in a sort meta-object and a progress computation object, just like in the PAC-Amodeus and Linguistic architectures. The model-view-controller triad for the progress is relative simple: the model contains the progress percentage, the view shows it as a bar on the screen, and the controller handles the 'Cancel' button.

3.2.5 Higgens UIMS

In the Higgens UIMS, the Find CD dialogue window can be interpreted as a view on the CD Collection. For the application data model the class diagram of the functional core can be used.
We will focus on the suggestion list functionality, which is also a view on the CD Collection. A Suggestion List View Node is associated with the CD Collection object. This view node can contain a list of artist names that match the user's input. It is also possible to generate this list each time it is needed from the functional core. A specific Artist Name View Node can be associated with each CD object. This view provides an interface to the artist name of a CD (irrespective of the actual representation of the name). It is possible to do without these view nodes and let the Suggestion List View Node retrieve the information from the CD objects. This alternative will have better performance (both memory and time) but the Suggestion List View Node will then depend directly on the CD objects.
The Suggestion List View Node is linked to two presentation objects, a text edit box and a list. The whole structure is shown in the figure below. Rectangles represent objects, edges represent dependencies between objects.

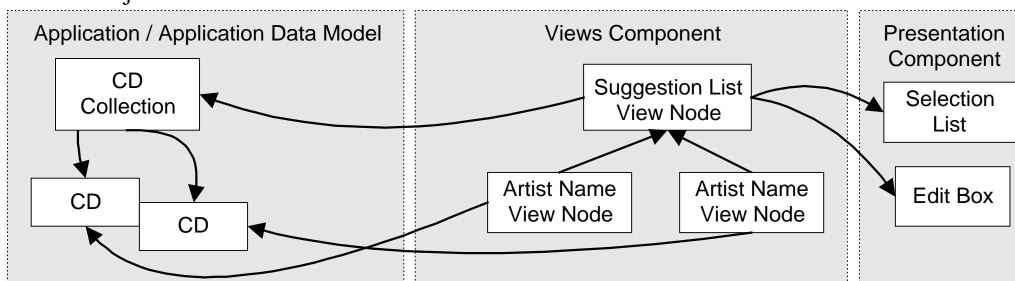

Figure 19. Higgens model of the global structure of the suggestion list

In the figure no distinction is made between the objects of the application and the interface to these objects provided by the application data model. The meta model of the application is not used here and it is not shown.
I will explain the dynamics of the model described above. If the user enters characters in the edit box, the edit box sends this information to the Suggestion List View Node. This view node determines which artist names match the user's input and puts these names in the selection list widget. If a string in the selection list is selected, the Suggestion List View Node sends the selected string to the edit box widget.
The suggestion list can be modelled well using the Higgens UIMS.

Modelling the progress indication using the Higgens UIMS is analogous to the PAC-Amodeus case. The Application Data Model provides the needed information about the sorting algorithm and the sorting process, e.g. through a sorting meta object. There is a Progress Viewing Node that is linked to the sorting meta object and which computes the percentage. The percentage is displayed using appropriate presentation objects.

3.3 Extensions and changes to the CD system

In this section, I will describe a number of possible extensions and changes to the CD database system and I will analyse what impact they have on the models that were constructed in the previous section. This gives an indication of the adaptability of the constructed models. The following extensions and changes will be described:

1. introduction of distribution and concurrency;
2. adding a picture of the artist to the CD system;
3. adding exception handling;
4. changing the sorting algorithm;
5. changing the type of progress indication.

Note that I will not work out all proposed extensions for all architectures presented earlier.

3.3.1 Introduction of distribution and concurrency

The first extension I propose is the introduction of distribution and concurrency aspects. The CD database system is extended to a multi-user system in which several users can access and update the CD information at the same time. The specific change I will focus on is the fact that while one user is interacting through the Find CD dialogue, a second user change the CD collection by adding or removing CDs. The suggestion list should reflect these changes immediately to prevent the user from picking an artist that does not exist any more.

The system should be modified to handle changes in the functional core and to update the interface components. This can be accommodated by applying the Observer design pattern to the involved components [Gamma et al., 1995]. In the Observer pattern, a specific object has the role of 'subject' that is observed by other objects. The subject is loosely coupled to its observers and it notifies them if the subject changes. The observing objects register themselves with the subject and they know how to handle the notifications of the subject.

Application of the Observer design pattern to the CD system amounts to the following: the CD Collection class becomes a 'subject' and other objects can register themselves as observers of the CD Collection object; if the contents of the CD Collection changes, it notifies its observers.

PAC-Amodeus

In the PAC-Amodeus design (alternatives 1.b and 2), the 'Artist Name Input' agent registers itself as an observer of the CD Collection. The sequence diagram shown in the figure below illustrates the process of change notification.

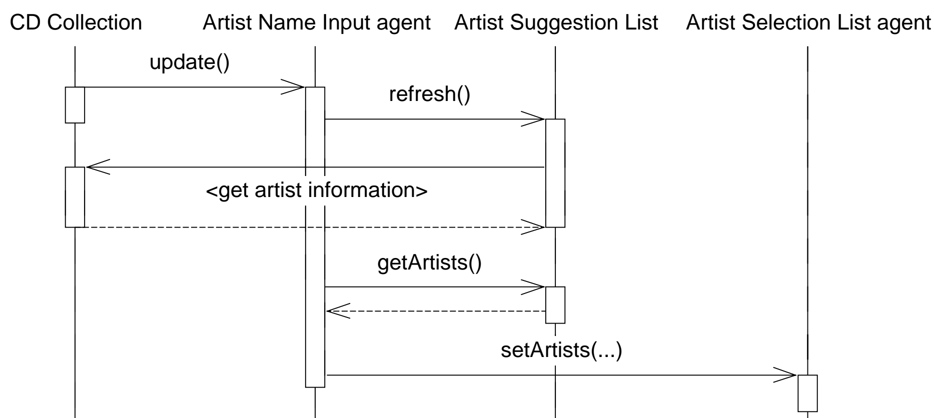

Figure 20. Sequence diagram of the change notification process

If the CD Collection changes, it sends a notification to the Artist Name Input agent (the `update()` message). This agent asks the Suggestion list to refresh itself; the suggestion list does so by retrieving the appropriate information from the CD Collection (shown by the '<get artist information>' edge). Finally,

the Artist Name Input agent retrieves the list of artist names from the suggestion list object and passes it to the Artist Selection List agent (`setArtists(...)` message).

Linguistic model

In the linguistic model of the CD system, the Find CD Dialogue object registers itself as an observer of the CD Collection. If the latter gives a change notification, the Find CD Dialogue object asks the suggestion list to rebuild itself.

Model-View-Controller

In the MVC model, the Artist Suggestion List Model registers itself as an observer of the CD Collection and rebuilds the suggestion list if necessary.

Higgens UIMS

If a CD object has its own Artist Name View Node, this view node will be notified of the change. The view node will notify the corresponding Suggestion List View Node, which will update the suggestion list.

Conclusion

Concluding, adding distribution and concurrency as described above can be handled well in the different user interface architecture without large costs. Some minor modifications are needed to the functional core and user interface objects.

3.3.2 Adding a picture of the artist

I propose to extend the concept of 'artist'. An artist does not only have a name, but also a picture. If the user selects an artist from the list of suggestions, the picture of the artist is shown in the dialogue window. This change affects both the application semantics, which should be extended to include a picture of the artist, and the interface, which should be extended to display the picture.
The functional core is extended with an 'Picture' object, which has to be associated with artists. Artists are no longer represented by character strings, but by an Artist class that has a name attribute and that contains an Picture object. The CD class is extended with an association to the Artist class. The changes in the class diagram of the functional core are illustrated in the figure below.

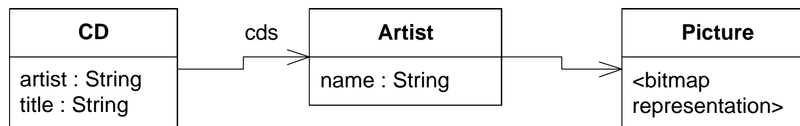

Figure 21. Modified functional core

PAC-Amodeus

I will describe the changes needed to alternative 2 of the PAC-Amodeus model of the Find CD dialogue. First, the 'Artist input' agent is extended with a subagent for displaying the picture (the 'Artist Picture' agent). This agent uses as presentation component e.g. a canvas or bitmap object, depending on what components are available to display images.
The 'Artist input' agent will co-ordinate the displaying of the picture when an artist name is selected in the suggestion list. The sequencing part of this change is not really a problem, it is just an extra rule added to the agent's control facet. To actually retrieve the picture however, the agent will need to know the corresponding Artist object. There are a number of alternatives to implement these changes:

1. the domain adapter component (or the functional core) provides an extra service to retrieve the picture given a name of an artist;
2. the 'Artist input' agent retrieves the Artist object corresponding to the selection of the user and asks this object for the picture.

Alternative 1 means more communication and coupling between functional core, domain adapter, and dialogue component. Alternative 2 means that the dialogue component needs intimate knowledge of the objects of the functional core. This is exactly the trade-off between coupling and cohesion that Hartson describes in [Hartson, 1989].
A second observation is that the 'picture of the artist' as a concern is distributed over several components. This decreases the adaptability of this concern: if it changes[4], modifications will have to be made to

[4] An example is a change from a bitmap representation to vector graphics representation.

several components. Changes in the user interface aspects of the artist picture influence the application semantics aspects and the other way around.
A third observation is that the picture object will need an interface to communicate with the user interface objects.
The figure below illustrates the modifications that are needed. The objects that have been modified or added have been shaded, bold edges represent modified or new dependencies.

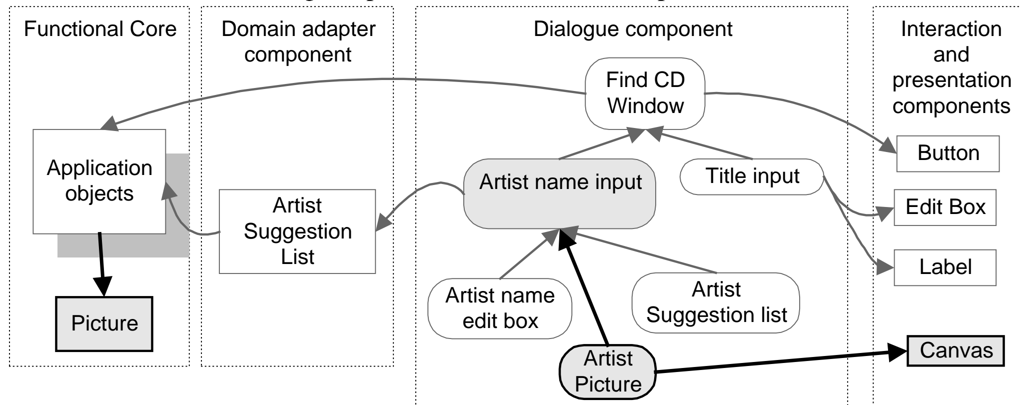

Figure 22. Modified PAC-Amodeus model

Linguistic model

In the linguistic model, the following changes and additions are needed. A lexical object (e.g. a Bitmap or a Canvas) is added. An object representing the artist picture is added to the syntactic layer. The object that represents the Find CD Dialogue window is extended with the responsibility of getting the picture from the right artist object. This is depicted in the figure below (modified objects are shaded, modified and new dependencies are bold).

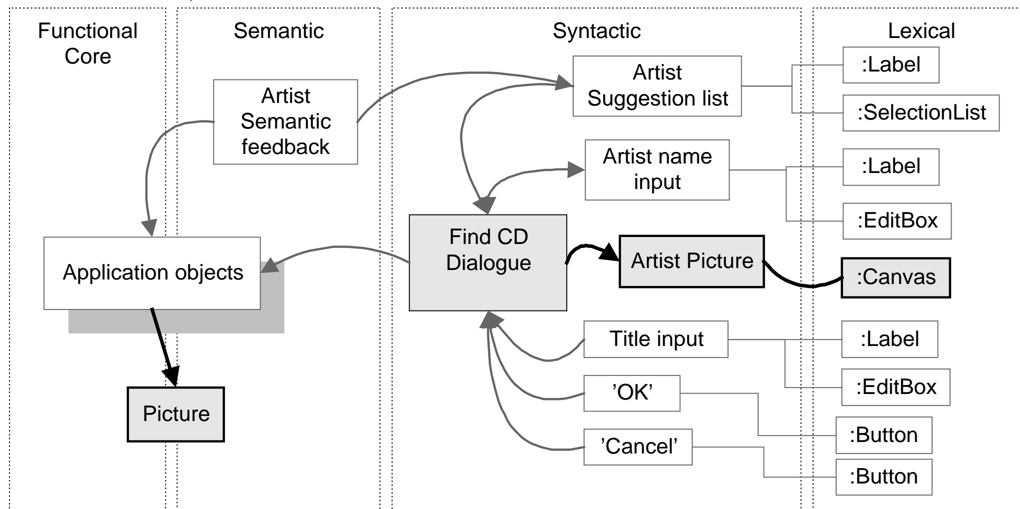

Figure 23. Modified Linguistic model

3.3.3 Adding exception handling

The issue of handling exceptions is actually a separate field of research with a number of open issues. Combining object oriented modelling and exception handling results in a number of anomalies, see e.g. [Miller, Tripathi, 1997]. The relevancy of exception handling for this case study is that exceptions tend to transcend the different components of user interface architectures.

An extension of the CD system augmented with a picture of the artist, as described in above, would be to retrieve the picture from a remote server from the Internet. This will introduce the possibility of distribution related exceptions: e.g. the server can be unavailable or the network connection can be too slow to show the picture immediately. Such exceptional cases originate in the functional core and one would like to abstract from them in the user interface, because distribution aspects should be transparent

to users. This is however not always possible, because if an exception occurs, the result will be visible to the user (or rather invisible, as the picture will not appear).
Another source of potential exceptional situations is concurrent access of the CD system by multiple users. This requires a transaction mechanism, which will introduce its own exceptions, e.g. a user wants to change the information on a certain CD, but he or she cannot perform the changes, because another user is accessing that particular CD.
Exception handling and giving appropriate messages can be seen as a special instance of (semantic) feedback. The problem is that the exceptional situation occurs in the functional core. The exception has to be interpreted and translated for other components of the system, and ultimately for the user, who does not think in terms of distribution but, in this case study, in terms of CDs. Concepts like distribution and concurrency, which should in general be transparent, suddenly become visible in the case of an exception.
In this case study, I will not work out the issue of exception handling for the different user interface architectures. Note that the specific differences between user interface architectures based on this separation do not seem to be relevant for exception handling.

3.3.4 Changing the sorting algorithm

A possible change regarding the progress indication is the use of a different sorting algorithm. A different algorithm has a different structure and different information is needed to compute the progress.
The new sorting algorithm needs to be extended with probes and the way the progress indicator computes the progress from the probe information needs to be changed. If there are several sorting algorithms available, each one with different advantages and disadvantages, it could even be desirable to change the algorithm at runtime[5].
Changing the sorting algorithm means that both the hooks or probes that extract runtime information from the sorting process and the structure of the algorithm are different. The progress computation will have to be modified to use this new information.
Changing the sorting algorithm only affects the algorithm itself, the sort meta-object, and the progress computation. If the sorting algorithm changes, the associated sorter meta-object also has to change if the new sorting algorithm has different run-time indicators of progress. The progress computation has to change if the interface of the meta-object changes and/or if the structure of the sorting algorithm changes. Other user interface components are not affected, therefore I will not elaborate on the changes required to the specific architectures presented above.

3.3.5 Changing the type of progress indication

Currently the progress indicator gives an indication of the percentage of work completed. This could be changed to an indication of the percentage of time that has been completed. Note that in general it is harder to compute a time percentage.
It would be nice if a generic solution for progress indication could be created that could be 'plugged' onto different algorithms. It is not desirable to extend each (sorting) algorithm with an interface for progress indication, because the interface and the implementation of the algorithm will have to be modified each time a different (progress) indicator is needed.
If the type of progress changes, the progress computation has to change. The sorter meta-object has to be changed if it does not offer the information needed to compute the new type of progress. If no reflection techniques are available, the sorting algorithm may have to be modified.
Note that the interface of the progress computation could remain unchanged, as time progress is still a percentage. The dialogue components are not always affected by a different type of progress indication (apart from a different label in the dialogue window).
Changing the type of progress in a radical way may affect the interface for the progress computation.
In summary, a different type of progress indication can lead to propagation of changes from the interface representation of the progress to the sorting algorithm.

3.4 Implementation issues

In this section, I will look at how the different models described in the previous sections can be made more concrete and how they can be mapped onto an implementation environment.
Traditionally, programming languages provide basic input/output through specific functions and libraries (e.g. the `stdio.h` library of C), or through built-in statements (e.g. the `read` and `write` statements in Pascal). In the C++ language, I/O based on streams has been provided. These are all low-level and often text or file oriented interaction mechanisms.

---

[5] Cf. the Strategy design pattern described in [Gamma et al., 1995].

Current programming languages, tools, and software development environments offer additional libraries or frameworks for user interfaces (usually for graphical user interfaces). They often prescribe, implicitly or explicitly, a specific model or architecture. The user interface frameworks of miscellaneous Smalltalk environments for instance are based on the model-view-controller architecture.
Visual GUI tools in software development environments like VisualWorks Smalltalk and IBM VisualAge for Java, usually generate a part of the user interface code in a specific way. This generated code can be quite hard to understand and modify manually, so software developers are often constrained to the specific way in which the user interface architecture has been worked out in the development environment.
Version 1.1 of the Java language and its libraries offers some freedom of choice of architecture. What is fixed is the AWT (Abstract Windowing Toolkit) library that offers abstract objects representing different user interface components. The AWT library hides the details of the underlying windowing system. In terms of the PAC-Amodeus model, the AWT library implements the interaction toolkit and presentation components. It uses a specific event handling mechanism, but does not specify how these events have to be handled ([JDK 1.2]; [Deitel, Deitel, 1998]). The Java language version 1.2 contains the Swing library, which provides an interface model that is at a higher level of abstraction.
The user interface architectures that I have described in this case study are models at a high level of abstraction. This means that there is a gap between a user interface architecture and its implementation in a particular programming language or development environment. Several design decisions have to be made before the architecture has been transformed into an executable model.

4 Evaluation

In this section, I will present an evaluation of the user interface architectures that I have used, an evaluation of applying the architectures to the case, and an evaluation of the problems of semantic feedback and adaptability that I have encountered.

User interface architectures

The PAC-Amodeus architecture allows different solutions satisfying the architecture. The PAC agent model allows a lot of flexibility. This can be both positive and negative. To constrain the space of possible solutions, in [Coutaz, 1990] and [Coutaz, Nigay, Salber, 1995] a number of rules (heuristics) are stated for constructing the hierarchy of agents. Buschmann et al. also provide a number of restrictions on the model, e.g. that the agent hierarchy should be kept shallow [Buschmann et al., 1996].
An important problem of the PAC-Amodeus architecture is in my opinion that the control facets of the agents tend to become complex and very situation-specific. The control facets encapsulate specific co-ordinated behaviour[6], encapsulating the dynamic behaviour of the interaction. The problem is that there is no framework or set of design rules for creating control facets. As a result, the agents are constructed in an ad-hoc manner and become less adaptable and reusable. The dynamics of the interaction, and thus the control facets, is generally prone to change, and modifying the control facets can become difficult and expensive.
The domain adapter suffers from similar adaptability problems. If specific functionality is moved from the dialogue agents to the domain adapter, the agents tend to become more generic and reusable, but the reusability of the domain adapter decreases. This component acts like a kind of glue and is affected by changes in both the functional core and the dialogue component.
The literature on user interface architectures offers in general few rules, heuristics or other ways of guidance to make the components of an interactive software system adaptable and reusable.

Application of the user interface architectures

The different architectures provide a way of classifying different concerns of the user interface and they define, to some extent, the coupling between these concerns. The concerns addressed by the different architectures are however similar. This is not a surprise, as most models are based on the separation of user interface and application and have been derived from the classic Seeheim model. As a result, the designs according to the different architectures that I have sketched are similar, and the same problems occur in the different architectures.
The solutions that I have sketched in the case study are not the optimal solutions. I believe however that the essential problems caused by semantic feedback will occur anyway. It is not clear e.g. where the functionality of creating the suggestion list should be placed. This concern belongs both to the application and to the user interface. It is a piece of functionality that exists for the purpose of the interface. Different

[6] The control facet can be seen as an instance of the Mediator design pattern described in [Gamma et al., 1995].

alternatives are possible that form a trade-off between mixing of concerns and high coupling between components.

Adaptability problems

In interactive software systems, the semantics of the interface and the application can become highly coupled. Different aspects of the user interface and the functionality are often not independent, but they affect each other in nonlocal ways (see e.g. [Gellersen, 1994]). In the case of semantic feedback, user interface aspects and application aspects are tangled, and they cannot be separated cleanly[7]. It requires extensive and frequent use of semantic information [Dance et al., 1992]. The Higgens UIMS can cope reasonably well with semantic feedback, but at the cost of high coupling between the view nodes and the application objects.

The tangling problem manifests itself as follows. Some concerns in interactive systems are hard to fit in when applying a separation-based user interface architecture. In the section on the 'find CD dialog window' for instance, a piece of functionality was needed to construct the list of suggestions. It was not clear where this piece of functionality should be located, because it has both user interface and functionality related aspects. Wherever it is placed, some of mixing of concerns takes place. The functionality of the suggestion list 'emerges' in the context of the interactive dialogue.

The picture of artist is another example of a concern that derives its reason for existence from the user interface. It has both application and interface aspects. When this concern was added, modifications were needed to both user interface and application components. The concern is scattered over the different components that result from applying a user interface architecture.

In the case of the progress indicator and the PAC-Amodeus architecture for example, it is hard to tell which aspects of progress indication belong to the functional core, which aspects belong to the dialogue and which aspects belong to the domain adapter component. The progress computation itself is a piece of functionality that only exists for the user interface, while it depends on the internals of the functional core.

The tangling phenomenon becomes problematic in the context of maintenance and adaptation of interactive software. It results in high coupling between components, in low cohesion of components and sometimes in duplication of concerns and creation of implicit dependencies. Concerns that are tangled become hard to adapt. The adaptability of the concerns of 'picture of the artist' and 'sorting progress indication' for instance is problematic: if one of them changes (e.g. the picture of the artist is changed from a bitmap representation to vector graphics representation), several components will have to be modified (problem of scattered changes). Changes to the user interface affect the application semantics and the other way around.

The tangling problem is in my opinion a fundamental problem of user interface architectures based on the separation of application and user interface. It is therefore not sufficient to introduce an extra layer or an extra component, because this does not tackle the essence of the problem. The Domain Adapter component of the PAC-Amodeus architecture for example tries to alleviate the problem by decoupling different components. It acts as a kind of glue between components and 'difficult' functionality as described above can be put in it. This happens however in an ad-hoc and unstructured (and therefore hard-to-adapt) manner. The adapter component tends to become a repository for components that the designer was unable to fit in the interface architecture.

Limitations and open issues

The CD registration system described in this report is a very small system. A more realistic example is needed to illustrate the problems of the separation of application and user interface. Other issues for further research are:

- Are there architectures for interactive software that can cope better with functionality like semantic feedback and that do not have the adaptability problems described in this report;
- What alternative (software engineering) views on interactive software can be useful;
- Are there other user interface concerns besides semantic feedback, that exhibit adaptability problems.

---

[7] This can be interpreted as user interface and application concerns *crosscutting* each other. This problem is similar to the problems addressed by the research on aspect-oriented programming [Kiczales et al., 1997].

5 References

[Andersen, 1997]
Peter Bøgh Andersen, *A theory of computer semiotics: semiotic approaches to construction and assessment of computer systems*, Updated ed., Cambridge etc., Cambridge University Press, 1997

[Bass, Clements, Kazman, 1998]
Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998 (SEI Series in Software Engineering)

[Buschmann et al., 1996]
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *Pattern-Oriented Software Architecture – A System Of Patterns*, John Wiley & Sons, 1996

[Coutaz, 1987]
Joëlle Coutaz, "PAC, an Object Oriented Model for Dialog Design", in: H.-J. Bullinger, B. Shackel (eds.), *Human-Computer Interaction - INTERACT'87, proc. of the 2nd IFIP conference on HCI*, 1987, pp. 431-436

[Coutaz, 1990]
Joëlle Coutaz, "Architecture Models for Interactive Software: Failures and Trends", in: G. Cockton (ed.), *Engineering for Human-Computer Interaction*, Elsevier Science Publishers B.V., North-Holland, 1990, pp. 137-153

[Coutaz, Nigay, Salber, 1995]
J. Coutaz, L. Nigay, D. Salber, "Agent-Based Architecture Modelling for Interactive Systems", in: P. Palanque, D. Benyon (eds.), *Critical Issues in User Interface Engineering*, Springer-Verlag, London, 1995, pp. 191-209

[Coutaz, 1997]
Joëlle Coutaz, "PAC-ing the Architecture of Your User Interface", in: *Design, Specification and Verification of Interactive Systems*, Proceedings of the Eurographics Workshop (DSV-IS'97), Springer Verlag, 1997, pp. 15-32

[Cunningham, 1995]
Ward Cunningham, "The CHECKS Pattern Language of Information Integrity", in: James O. Coplien, Douglas C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley Publishing Company, 1995, pp. 145-155

[Dance et al., 1992]
J.R. Dance, T.E. Granor, R.D. Hill, S.E. Hudson, J. Meads, B.A. Myers, A. Schulert, "The Run-Time Structure of UIMS-Supported Applications", in: E. Edmonds (ed.), *The Separable User Interface*, 1992, pp. 211-226

[Deitel, Deitel, 1998]
H.M. Deitel, P.J. Deitel, *Java – How to Program*, 2nd ed., Prentice Hall, 1998

[Ecklund et al., 1996]
Earl F. Ecklund, Jr., Lois M.L. Delcambre, Michael J. Freiling, "Change cases: use cases that identify future requirements", in: *Proceedings OOPSLA'96*, 1996, pp. 342-358

[Edmonds, 1992]
Ernest Edmonds, "The emergence of the separable user interface", in: E. Edmonds (ed.), *The Separable User Interface*, Academic Press, 1992, pp. 5-18

[Encarnação, 1997]
Chapter 2 of: L.M. Encarnação, *Concept and Realization of Intelligent User Support in Interactive Graphics Applications*, Ph.D. thesis, Fakultät für Informatik der Eberhard-Karls-Universität, Tübingen, 1997

[Fayad, Cline, 1996]
Mohamed Fayad, Marshall P. Cline, "Aspects of Software Adaptability", in: *Communications of the ACM*, vol. 39, iss. 10, October 1996, pp. 58, 59

[Fenton, Pfleeger, 1996]
N.E. Fenton, S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed., London

etc., Thomson, 1996

[Foley et al., 1990]
Chapter 9 of: James D. Foley, Andries Van Dam, Steven K. Feiner, John F. Hughes, *Computer Graphics – Principles and Practice*, Addison-Wesley Publishing Co., Reading, MA, 1990

[Gamma et al., 1995]
E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[Gellersen, 1994]
Hans-W. Gellersen, “Support of User Interface Design Aspects in a Framework for Distributed Cooperative Applications”, in: R.N. Taylor, J. Coutaz (eds.), *Software Engineering and Human-Computer Interaction. Proceedings*, LNCS 896, 1995, pp. 196-210

[Görner, Vossen, Ziegler, 1992]
Claus Görner, Paulus Vossen, Jürgen Ziegler, “Direct Manipulation User Interface”, in: M. Galer, S. Harker, J. Ziegler (eds.), *Methods and Tools in User-Centred Design for Information Technology*, Elsevier Science Publishers, 1992, pp. 237-279

[Gram, Cockton, 1996]
Christian Gram, Gilbert Cockton (eds.), *Design Principles for Interactive Software*, Chapman & Hall, 1996

[Green, 1985]
M. Green, “Report on Dialogue Specification Tools”, in: Günther E. Pfaff (ed.), *User Interface Management Systems*, Springer Verlag, 1985, pp. 9-20

[Hartson, 1989]
Rex Hartson, “User-Interface Management Control and Communication”, in: *IEEE Software*, vol. 6, iss. 1, January 1989, pp. 62-70

[Hoffner, Dobson, Iggulden, 1990]
Yigal Hoffner, John Dobson, David Iggulden, “A New User Interface Architecture”, in: G. Cockton (ed.), *Engineering for Human-Computer Interaction*, 1990, pp. 113-136

[Hudson, King, 1988]
Scott E. Hudson, Roger King, “Semantic Feedback in the Higgens UIMS”, in: *IEEE Transactions on Software Engineering*, vol. 14, iss. 8, August 1988, pp. 1188-1206

[JDK 1.2]
Java™ Development Kit Documentation, JDK™ 1.2

[Kiczales et al., 1997]
Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, John Irwin, “Aspect-oriented programming”, in: M. Aksit, S. Matsuoka (eds.), *ECOOP’97 proceedings*, Springer Verlag, 1997, pp. 220-242

[Krasner, Pope, 1988]
Glenn E. Krasner, Stephen T. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, in: *Journal of Object-Oriented Programming*, August/September 1988, pp. 27-49

[Maes, 1987]
Pattie Maes, “Concepts and Experiments in Computational Reflection”, in: *OOPSLA’87 Proceedings*, 1987, pp. 147-155

[Microsoft Developer Network]
Microsoft Developer Network, http://msdn.microsoft.com

[Miller, Tripathi, 1997]
Robert Miller, Anand Tripathi, “Issues with Exception Handling in Object-Oriented Systems”, in: M. Aksit, S. Matsuoka (eds.), *ECOOP’97 Proceedings*, Springer Verlag, 1997, pp. 85-103

[Myers, 1985]
Brad A. Myers, “The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces System Response Factors”, in: *Proceedings of ACM CHI’85 conference*, pp. 11-17

[Myers, Rosson, 1992]
Brad A. Myers, Mary Beth Rosson, "Survey on User Interface Programming", in: *Proceedings of SIGCHI'92: Human Factors in Computing Systems*, May 3-7, 1992

[Nielsen, 1993]
Jakob Nielsen, *Usability engineering*, Academic Press, 1993

[Pressman, 1992]
Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 3rd ed., McGraw-Hill International Editions, 1992

[Reynolds, 1997]
Carson Reynolds, "A Critical Examination of Separable User Interface Management Systems: Constructs for Individualization", in: *SIGCHI Bulletin*, Vol. 29, Iss. 3, July 1997

[Rosenberg et al., 1988]
Jarrett Rosenberg (moderator), Ralph Hill, Jim Miller, Andrew Schulert, David Shewmake (panelists), "UIMSs: Threat or Menace?", in: *CHI'88 Conference Proceedings*, Addison-Wesley Publishing Company, 1988, pp. 197-200

[Treu, 1994]
Sigfried Treu, *User Interface Design – A Structured Approach*, Plenum Press, NY, London, 1994, (Languages and Information Systems, Shi-Kuo Chang, series editor)

[Versendaal, 1991]
J.M. Versendaal, *Separation of the User Interface and Application*, Ph.D. thesis, Delft University of Technology, Delft, 1991

[Wiecha et al., 1989]
C. Wiecha, W. Bennett, S. Boies, J. Gould, "Generating Highly Interactive User Interfaces", in: *CHI'89 Proceedings*, 1989, pp. 277-282