Separable User Interfaces and Interaction Controls

Hanna Dobos

Master's Thesis

Software Engineering Study Line

November 2002

University of Jyväskylä

Department of Mathematical Information Technology

Author: Hanna Dobos

Contact information: Hanna.Dobos@web.de

Title: Separable User Interfaces and Interaction Controls

Työn nimi: Separoituvat käyttöliittymät ja käyttöliittymäoliot

Project: A Master's Thesis in Information Technology, Software Engineering study line

Page count: 78

Abstract: This thesis looks for an answer to the question: What should the architecture of an interactive system be like so that the user interface and the application can be separated? First some existing user interface architectures that support the separability of the user interface are introduced. Special attention is then paid to the most important part of the architecture concerning the separability, the interface between the user interface and the application. For the rest of this thesis, the focus is set on structures of interaction controls that support separability. The user interface architecture and the interaction control structure of an existing interactive system are introduced and evaluated. This interactive system was improved with the goal of making the user interface separable.

Suomenkielinen tiivistelmä: Tämä pro gradu -tutkielma etsii vastausta kysymykseen: Millainen tulee vuorovaikutteisen sovelluksen käyttöliittymäarkkitehtuurin olla, jotta käyttöliittymä ja sovellus voidaan erottaa toisistaan? Ensin esitellään muutama olemassa oleva käyttöliittymäarkkitehtuuri, jotka tukevat käyttöliittymän separoimista. Erityisesti huomiota kiinnitetään käyttöliittymän ja sovelluksen väliseen rajapintaan, koska se on tärkeimmässä asemassa separoituvuuden kannalta. Loppuosassa työtä pääpaino asetetaan separoituvuutta tukeville käyttöliittymäolioiden rakenteille. Erään olemassa olevan vuorovaikutteisen sovelluksen käyttöliittymäarkkitehtuuri ja käyttöliittymäolioiden rakenne esitellään ja arvioidaan. Tätä sovellusta kehitettiin tavoitteena tehdä käyttöliittymä separoituvaksi.

Keywords: Separable user interface, user interface architecture, HCI, UIMS, Seeheim, KBFE, Arch, MVC, PAC, PAC-Amodeus, Design pattern, Application interface, Interaction control

Avainsanat: Separoituva käyttöliittymä, käyttöliittymäarkkitehtuuri, HCI, UIMS, Seeheim, KBFE, Arch, MVC, PAC, PAC-Amodeus, suunnittelumalli, käyttöliittymärajapinta, käyttöliittymäolio

Acknowledgements

First I would like to thank my supervisor at S|4|M, Dr. Martin Fischer, for all his support. His comments and criticism repeatedly helped me to look at the topic from different viewpoints. I also want to thank all my colleagues at S|4|M for urging me on as I was writing this thesis.

I extend my thanks to Prof. Dr. Tommi Kärkkäinen from the Department of Mathematical Information Technology at the University of Jyväskylä for his support during this thesis and also earlier in my studies. I am grateful for the comments and suggestions that he and Assistant Prof. Jonne Itkonen made for the improvement of this work.

Very special thanks belong to my parents for believing in me and supporting me over the years of my education. Last but not least I would like to thank my husband Andreas for his understanding and assistance.

Contents

1 Introduction

The internal structure of user interfaces (UI) is becoming increasingly complex, as there is a continuous demand to have UI to support more and more devices, media, users, tasks and environments, with more and more dialogues active at the same time. These increasingly rapid technological changes are likely to significantly change the appearance and interaction possibilities of user interfaces.

One new, significant challenge for the user interfaces of interactive software systems is *ubiquitous computing*. The idea of ubiquitous programming is that computation will be embedded in many different devices: personal digital assistants (PDAs), cell phones, desk and wall-size computers, cars, microwave ovens, clothes etc. Virtually none of these devices work with the traditional input devices: keyboard and mouse. They use, for example, touch screens, stylus, speech, gesture, or handwriting recognition [28].

There is also an increasing need of end user customisation, as heterogeneity of end users rises: the users differ in age, personality and working style. Furthermore, the users change with time: when a user starts to learn a computer system, he needs more support than later, when he is familiar with the system.

All these changes require significant support from the underlying user interface architecture. Conventional graphical user interface techniques appear to be ill suited for some of the new kind of interactive platforms now starting to emerge, like ubiquitous computing devices having tiny and large displays. It will be important to have replaceable user interfaces for the same applications to provide different user interfaces on different devices for ubiquitous computing, and to support customizing. This encourages a return to the study of some techniques for device-independent user interface specification and implementation [28].

The *user interface separability* means that the user interface can be replaced or reused without any or with only little modifications in the application. Separable user interfaces have already been a subject of research for decades. The foundations on this area were laid in the 1970's and 1980's. For a long time the separability was not essential, because of the uniformity and unchangeability of the user interfaces.

Because of recent development in the user interface technologies during the last decade, the separable user interfaces became an active area of research again.

In the next sections 1.1 and 1.2 the development environment used in this work and the goals of this work are introduced.

After that some relevant terms and acronyms are defined in section 1.3. Chapter 2 deals with user interface architectures that support the separability of the user interface. This work restricts to architectures for single-user interactive systems. In chapter 3 the separability is handled from the application interface point of view. The task of connecting application data to the user interface is also discussed in chapter 3. Some semantic models that help separating the interaction controls on the user interface are introduced in chapter 4. In chapter 5 the theory introduced in the preceding chapters is put into practice: the user interface architecture and interaction control hierarchy of an existing intercative system are introduced.

1.1 S|4|M

S|4|M -Solutions for Media develops software for the use of television broadcasting companies. With help of the software-package corresponding to this work, S|4|Rights, all decision-relevant information is immediately available for the user. S|4|Rights consists of several modules, which as a whole cover the key business areas of a television broadcasting company concerning license management and program logistics: Licensing Rights Administration, Film and Series Administration, Material and Archive Administration and License Accounting.

1.1.1 The development environment in S|4|Rights

S|4|Rights is developed using Visual C++ and Microsoft Foundation Classes (MFC). The core of S|4|Rights-applications is the MFC application framework, a group of C++ classes in the MFC library that provides the essential components of an application for Windows. The application framework defines a skeleton of the application and supplies standard user interface implementations that can be placed onto the skeleton [27].

MFC shortens development time, makes code more portable and provides support without reducing programming freedom and flexibility. The MFC framework consist more or less of a few major classes that encapsulate a large portion of the Win32 application programming interface, application concepts (documents, views, application itself) and data-access functionality [27].

The key concepts in an MFC application are document, view, frame windows, document template, application object and thread objects. The *document class* specifies the data of the application. The *view class* is the user's sight of the application data. It controls how the user sees and interacts with the data in the document. It is possible to have multiple views for the document. Views are displayed inside the document *frame windows* [27].

A *document template* coordinates the creation of documents, views and frame windows. The *application object* controls all the objects above and specifies application behaviour. If the application creates separate threads of execution, *thread objects* are used to handle this case [27].

In a running application, these objects are bound together by commands and messages. Figure 1 below shows how the objects communicate with each other.

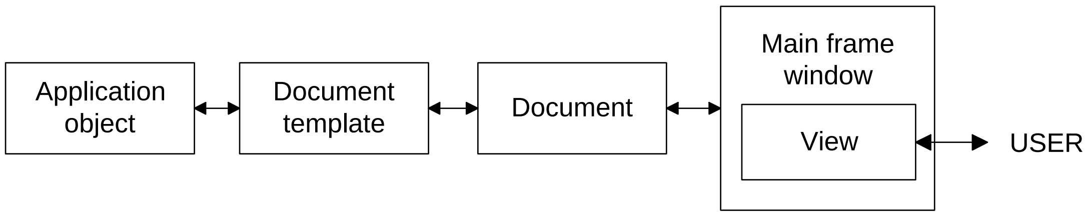


Figure 1: The MFC application framework [27].

The MFC library provides a large group of interaction controls used in the most windows applications. Because of the wide use, applying these interaction controls to the user interface makes it familiar and usable for the end users. The designers can use Visual C++ resource editors for creating user interface objects: menus, dialog boxes, interaction controls, icons and so on [27].

1.1.2 The current state of development

Over the years of development, the different modules of S|4|Rights have been developed more or less independently, causing individual solutions and repeated code within the modules. However, these modules share some object libraries and a common framework, based on the Microsoft Foundation Classes (MFC) framework.

Many steps have already been taken to achieve consistency within the modules of S|4|Rights. One example for this is the use of common base classes as interfaces for object hierarchies, like for the document classes or the views in the modules. These interfaces make the use of many common procedures for all the modules possible.

The base class for all MFC windows classes (views, dialogs, controls, etc.) is `CWnd`. The interaction controls are directly inherited from `CWnd` without a common base class [27].

The lack of interface for the interaction controls makes the resulting code sequences dependent on specific interaction controls. Changing the interface is therefore difficult: already small changes may require modification of many parts of the application code. Replacing the whole user interface by another user interface with another interaction style would require enormous amount of work and time.

1.2 The goals of this work

Todays computer systems must be prepared to accept the challenge of offering multiple communication possibilities for the end user of an application. This means especially that massive changes on the user interfaces of interactive software systems may be necessary.

The goal of this work is to study different user interface architectures that support the separation of the user interface and the application. A special focus is set on the structure of interaction controls that support separability.

The main issue of this work is to consider the question

1. What kind of architecture should an interactive software system have so that the user interface and the application can be separated?

Some further question to be treated are

2. How should the structure of interaction controls be build so that the user interface changes do not cause significant modifications to the business logic?

3. How should the application data needed for the communication with the user be bound to the user interface without disturbing the separability?

The hierarchy of the interaction controls should be easy to extend and modify without causing significant changes to the existing hierarchy. One practical goal of this work is also to develop a new navigation control for the S|4|Rights modules that utilizes the chosen structure for the interaction controls.

1.3 Terms and Acronyms

Agent

An agent is a specialised computational unit that has a state and is capable of initiating and reacting to events [8].

Application

The application is here used to describe those parts of the system that do not have to do with the user interface. It includes the semantic and logical intelligence of the software system. This part is sometimes also called computational software or business logic.

Application interface

Application interface is the interface between user interface and application. It plays a significant role in the separability of these parts in an interactive software system.

Architecture

Software architectures are used to structure the application in a proper manner, corresponding to the goals that are set for the application. In the architectural design the application is typically ordered in modules or layers, which all have their own tasks and responsibilities and are connected to each other in a reasonable way.

Design pattern

With the words of Alexander [3]:

> ”Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”.

Device independence

A characteristic of software that generates the same output regardless of the hardware involved. A device-independent program could, for example, issue the same command to draw a rectangle regardless of whether the output device was a printer, a plotter, or a screen display [27].

Framework

A framework is a set of cooperating classes that make up a reusable design for a specific class of software. It dictates the architecture of the application and defines the overall structure, its partitioning into classes and objects, how the classes and objects collaborate, and the thread of control [18].

HCI

HCI - Human-Computer Interaction is a discipline dealing with the design, evaluation and implementation of interactive software systems for human use, along with the study of major phenomena surrounding them [21].

Interaction control

An interaction control is a child window on the screen, acting as an interface for a variable. Its value can be manipulated by the user to perform an action or to display information. Interaction controls are also called widgets, interactors, interaction techniques, user interface objects (UIO), presenters, etc.

MFC

MFC - Microsoft Foundation Classes is a set of C++ classes that encapsulate much of the functionality of applications written for the Microsoft Windows operating systems [27].

Separability

Separability of a computer system means that parts of the system can be replaced or reused without any or with only little modifications. Changes in one part of the system do not cause changes in other parts, as long as both remain consistent with their common internal interface representation [20].

S|4|M

S|4|M - Solutions for Media is IT-company that develops software for the use of television broadcasting companies.

S|4|Rights

S|4|Rights is software package developed by S|4|M, consisting of several modules, which as a whole cover the key business areas of a television broadcasting company.

Toolkit

A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality [18].

UI

UI - User Interface (also called human-computer interface, the front end or the man-machine interface) is that part of a system that the user is in contact with either physically, perceptually or conceptually [26].

UIMS

UIMS - User Interface Management System is a software system that takes over the functions of managing the user interface, leaving the application program to get on with processing [11].

2 User Interface Software Architectures

Software architectures are used to structure the application in a proper manner, corresponding to the goals that are set for the application. In the architectural design the application is typically ordered in modules or layers that each have their own tasks and responsibilities and are connected to each other in a reasonable way.

Architectural design supports standardization and reuse of the modules, makes the dependencies of the modules local and improves the exchangeability of parts of the application. Furthermore, it enables the locality of changes and stability of interfaces. Splitting the application into modules also helps dividing the work of software designers, developer and testers. On the other hand it reduces the efficiency of the application [24].

Instead of designing the architecture of an application from the scratch it is possible to use application frameworks that dictate the architecture. The user interface for an interactive system can be build using a user interface builder. However, one problem with application frameworks and user interface builders is that they do not make explicit the connection between the functionalities they offer and their underlying architecture. The software architecture is lost in the resulting code so that the programmer must rearrange the architecture in order to reuse and extend the existing code appropriately. The user interface generators also tend to give the programmer a false sense of confidence that the software architecture is no longer an issue [8].

Modelling of the software architecture can significantly help in clarifying these problems. It provides the right insights, assists in making the right questions and offers general tools for organizing thoughts. The product of architectural modelling is a set of computational entities called *components* and their *connections* with each other. An architecture is mostly expressed in natural language complemented with a graph of labelled boxes and arrows between them (see, e.g., Figure 2) [8].

The steps taken in the architectural modelling process include identifying the conceptual pieces of the system, defining the structure

of the system (how the conceptual pieces relate to each other), allocating functions to the structure and describing the dynamic behaviour of the architecture. Finding the right balance between multiple sources of requirements is often a big difficulty in the process [8].

Developers of interactive systems must make difficult compromises in order to optimise the development processes and end products. The compromises are made among desirable, but sometimes conflicting, requirements and goals such as minimizing the future effects of changing technology and improving the runtime performance of a system. Other possible design criteria are quality of resulting user interface, reuse of code, complexity of specification, target system extensibility, time to market, compatibility with other systems, etc [1]. Techniques like separation of concerns and aspect-oriented programming are developed to help designers to cope with these multiple requirements (see, e.g., [29] and [9]).

In the next section 2.1 the human computer interaction (HCI) paradigm is introduced. It should always be taken into account when designing interactive systems. After that, in section 2.2, an early approach to the user interface part of an interactive system, the UIMS, is discussed, followed by some significant early user interface architectures in section 2.3. In section 2.4 two agent-based models for the user interface are introduced and, at last, in section 2.5 the benefits of design patterns in the architectural design are outlined.

A common approach for developing models for interactive systems is to examine the functionality of the system, decide that separating the user interface from the application is the most important design goal, and derive an architecture that supports this separation [1]. The Seeheim and KBFE models introduced in subsections 2.3.1 and 2.3.2 are examples of this. Although such a prescriptive model is desirable, it is very difficult, if not impossible, to define a model that fits to all types of interactive systems. The Arch and agent-based models introduced in subsections 2.3.3, 2.4.1 and 2.4.2 take a different approach. They do not propose a particular architecture, but instead examine the nature of data that is passed between the user interface and the application. The Arch model, for example, was tailored to satisfy the particular goal of minimizing the future effects of changing technology [1].

2.1 HCI -Human Computer Interaction

Human-Computer Interaction (HCI) is an important area concerning user interfaces. The goal of HCI research is to make user interfaces easier to learn and use. Design in HCI is more complicated than in many other fields of software engineering, because it is influenced by diverse areas such as computer science (application design and engineering of user interfaces), psychology (the application of theories of cognitive processes and the empirical analysis of user behaviour) and human factors (usability, learnability, memorability, helpfulness of user interfaces) ([6], [21]).

The developer's goal of making a complex system appear simple and sensible to the user is a very difficult and complicated task. The area of computer science HCI studies and develops the abstractions, techniques, languages, and tools to address this problem [6]. Even from this perspective, it is advantageous to frame the problem of human-computer interaction broadly enough to help practitioners avoid the classic pitfall of design divorced from the context of the problem [21].

Foley et al [16] state that there is a helpful analogy between user-computer dialogue and interpersonal communication. The language of the conversation should be the language of the user, not that of the computer or developer. It should be efficient, complete and easy to learn. Getting feedback, as well as being able to undo mistakes, are very important components of the interaction.

Foley et al [16] introduced a model for the design of interactive user-computer conversations, where they separate the design into distinct levels: the conceptual, semantic, syntactic, and lexical levels. The *conceptual design* is the definition of the key application concepts the user must master. Hence, it is also called as the *user model* of the application. The conceptual design typically defines objects, relationships between the objects, and operations on the objects.

The *semantic design* specifies detailed functionality: what information is needed for each operation on an object, what semantic errors may occur and how they are handled, and what are the results of each operation. Finally, the *syntactic design* defines the sequence of inputs and outputs and the *lexical design* determines how input and output is actually formed from the available hardware primitives.

2.2 UIMS - User Interface Management Systems

The literature concerning interactive systems seems not to be able to agree on the definition of user interface management system (UIMS). Some sources (like [14] or [22]) see that UIMS is one term for the user interface part of a computer system, meaning that an interactive software system consists of two parts: application and UIMS. Other sources (like [28] or [17]) have a more restricted meaning for UIMS: they say that an UIMS has an analogy to database management systems (DBMS). A DBMS hides low-level concepts such as disks or files from the developer, automates many previously very tedious tasks and allows independence from low-level details. Similarly, a UIMS should abstract details of input and output devices, providing standard or automatically generated implementations of interfaces.

The second explanation for UIMS has been much criticized already in the early days of its existence. Myers et al [28] argue that for every user interface, it is important to be able to control the low-level characteristics of the interaction that the UIMS tries to hide from the designer. Manheimer [25] states that because a UIMS package is based on an abstraction of contemporary user interfaces, it will become obsolete as advances in user interface technology occur. Myers et al [28] confirm this critic by saying that the standardization of the user interface elements made the need for abstractions from the input devices mostly unnecessary.

The current user interface development shows that the details of the interaction must be available for the designer, but on the other hand, it would be useful to be able to separate the details from the user interface for improving changeability. This means that some level of abstraction in form of an interface is needed anyway.

According to the first explanation UIMS is only an old term for the user interface. Today it is rarely used in the literature. The rest of this subsection is valid for both definitions of UIMS.

A goal of UIMS is to enhance efficient development of high-quality user interfaces.

To accomplish this, the application and the user interface are logically separated from each other. This separation allows the specialists to develop the user interface and the application independently, promotes consistent appearance across modules, and allows application functions to be accessed or combined in new ways, promoting reuse ([22],[25]).

The responsibility of an UIMS is to deal with all tasks related to the user interface, including visual representation and the control of the communication between the user and the user interface. Most UIMS contain high-level tools for user interface specification, such as screen builders and dialogue editors [25], but they are not only interaction technique toolkits, providing also additional functionality in implementing the user interfaces [17].

Most UIMS provide some means of defining allowed sequences of user actions and may, in addition, support help and error messages, macro definitions or user profiles. UIMS can increase programmers' productivity and speed up the development process [17].

2.3 Some important early models

In this section some fundamental architectural models are introduced. The Seeheim model considered in subsection 2.3.1 was the very first model providing a functional decomposition for the UIMS technology. Its components correspond roughly to the semantic, syntactic and lexical levels of Foleys language model introduced in section 2.1.

The KBFE-Architecture in subsection 2.3.2 delivers a model for attaching the application data to the user interface for providing appropriate semantic feedback and end user support.

In the Arch model, introduced in subsection 2.3.3, the level of abstraction of the user interface partition is still raised. The model insulates the dialogue control component in the Seeheim model from its functional partners and promotes therefore especially reuse of components and portability of the user interface [8].

2.3.1 Seeheim

An important event in the development of separable user interface architectures was the workshop held in 1983 in Seeheim. One group in the meeting, consisting of J. Derksen, E. Edmonds, M. Green, D. Olsen and R. Spence, developed an abstract model of an user interface, widely known as the *Seeheim model* [12]. The model is illustrated in Figure 2.

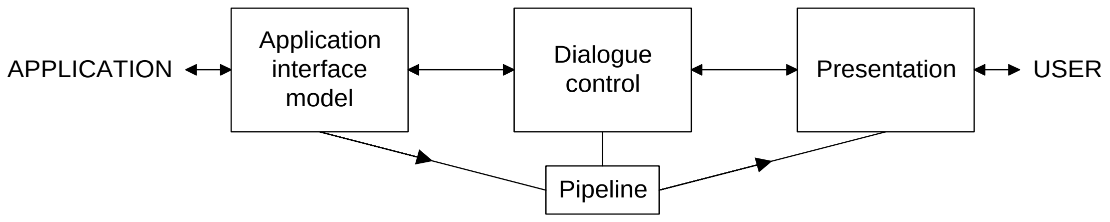


Figure 2: The Seeheim Model of an user interface [19].

This model presents the logical components that should appear in an user interface. Each component has different functions that are next introduced.

The *presentation* component is responsible for the external representation of the user interface. This component generates the images that appear on the display screen, and reads the physical input devices, converting the raw input data into the form required by other components in the user interface [19].

The *dialogue control* component defines the structure of the dialogue between the user and the application program. The user makes requests through the presentation component and supplies data to the application. Similarly, the application generates requests for data and answers to the user requests. The dialogue control component must channel these requests to the appropriate routines in the application, or correspondingly, to appropriate parts of the presentation component. The actions performed by the dialogue component usually depend on the context of the dialogue; therefore, the component must be able to handle dialogue states and state changes [19].

The *application interface model* is a representation of the application from the user interface viewpoint. This component defines the interface between the user interface and the rest of the application: it contains a description of the application data structures that are of interest to the user interface and the routines the user interface can use to communicate with the application. It also contains constraints on the use of the application routines. This allows the user interface to check the semantic validity of the user input before application routines are invoked [19].

The box under the dialogue control in Figure 2 represents a "pipeline" between the application and the presentation component. According to Green [19], a direct connection between these two components could be useful in some cases. One example of this are the user interface picks. When user picks an object on the user interface, the presentation component knows the coordinates of the pick, but it does not know how to relate these coordinates to contents of the application data structure that is currently displayed on the screen. Some correlation mechanism is needed. Another example is the flow of output data from the application to the presentation component. Since dialogue control does not need to process the data, it could be transferred directly to the presentation component. Dialogue control establishes the pipeline, but after that it does not take part in the information transfer [19].

Dance et al [10] introduce a structure for the user interface similar to the Seeheim model, but where the dialogue control and application interface are melted together to a dialogue manager that contains a part called semantic support component. This structure is shown in Figure 3. Like in the Seeheim model, the primary purpose of the dialogue manager is to provide a higher level of abstraction for interaction services.

The purpose of the semantic support component is to provide information for semantic operations such as feedback, default values and error checking and recovery. Furthermore, it supports modifications on the user interface based on the application state, like changing menu contents.

The semantic support component is similar to the application interface model proposed in the Seeheim architecture. However, instead of being a passive interface, Dance et al view the semantic support component as an active component of the dialogue manager that not only specifies the interface to the application, but also defines how the semantics of the application are incorporated into the user interface.

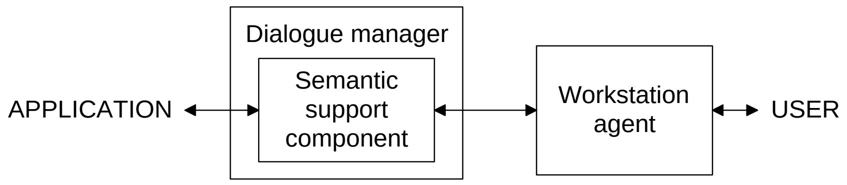


Figure 3: A model for an user interface proposed by Dance et al [10].

Dance et al [10] discuss different possibilities to couple the semantic support component with both the application and dialogue manager, but can't give a definitive answer to that problem. One possibility is introduced in the next section.

2.3.2 KBFE - Architecture

The Knowledge-based Front End -Architecture (KBFE) has been developed to support the construction and maintenance of knowledge-based front ends for existing software systems. The major objective is to enhance the usability of the system by providing both improved user interfaces and knowledge-based support to the end user. The KBFE-architecture provides an approach to the integration of diverse software systems and to the provision of support for their use [13].

The central component in the KBFE architecture is Harness (see Figure 4), which is responsible for the control and management of the user interface and all of the communication between the various front-end modules and the back-end software system. The Harness can be used to integrate an unspecified number of facilities running on various machines with different environments, using a shared user interface [13].

The KBFE-architecture is an extension of the Seeheim model, which is, in effect, subsumed within the Harness. The application interface in Seeheim model contains an explicit model of the underlying application in order to provide a clear interface between the front and back ends. The messages employed in the KBFE-architecture are introduced as an approach to a key issue of specifying a complete and consistent notation in which a specific interface can be defined [13].

In addition to the three Seeheim user interface layers, the KBFE contains an unspec-

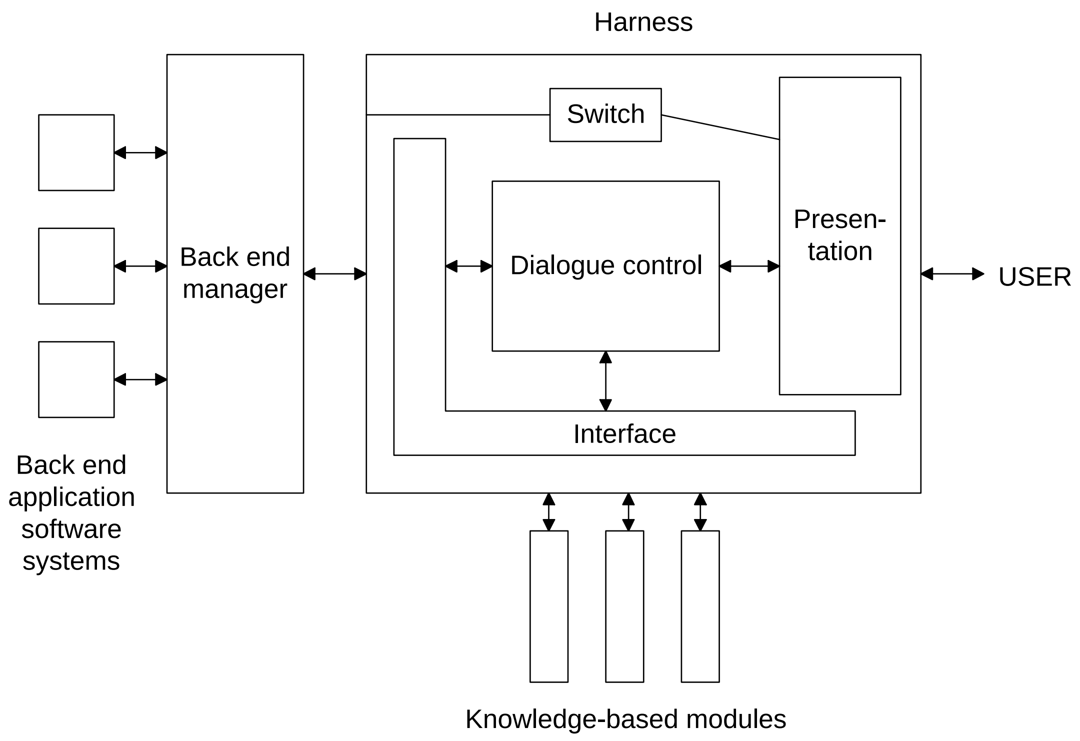


Figure 4: The KBFE architecture [13].

ified number of knowledge-based modules, which provide the user with guidance and assistance. They can, for example, incorporate knowledge about the functions supplied in the application software as well as the ways in which they can or cannot be used in relation to particular problems. The key point is that whatever facilities the developer is willing to supply, the architecture allows them to be modularised appropriately and run as concurrent processes [13].

Besides Harness, another important component in KBFE is the back-end manager, which deals with the problem of front-end separability. The back-end manager handles the creation, management and use of back-end data objects. Its prime responsibility is that of mapping a software system -independent specification of a task into a form that can be used to actually execute the task using the particular software. The back-end manager can also use more than one software systems at the same time [13].

The different software processes in KBFE architecture communicate with each other using a single message-passing convention, through Harness. Each module has a link with the Harness and the communication can be routed through it. In providing this facility, the Harness maps between the abstract interaction objects of interest and the physical realizations of them that are meaningful to the user [13].

Similarly to the Seeheim model, the KBFE-architecture contains a high-bandwidth route from the back ends to the presentation layer of the interface. It passes through a "switch", also illustrated in Figure 4, which directs the data, under the direction of dialogue control, without having to pass through the various software components that together make up the full harness [13].

In the back-end manager, the interface between the user interface and the application is realized in terms of *tasks* and *actions*. A task is an application-independent specification of a back-end activity. It consists of the control and a group of one or more actions, for each back-end application whose functionality allows its implementation. An action can be seen as the control and a group of one or more back-end application commands forming a unit, which are perceived to be useful for building up tasks [30].

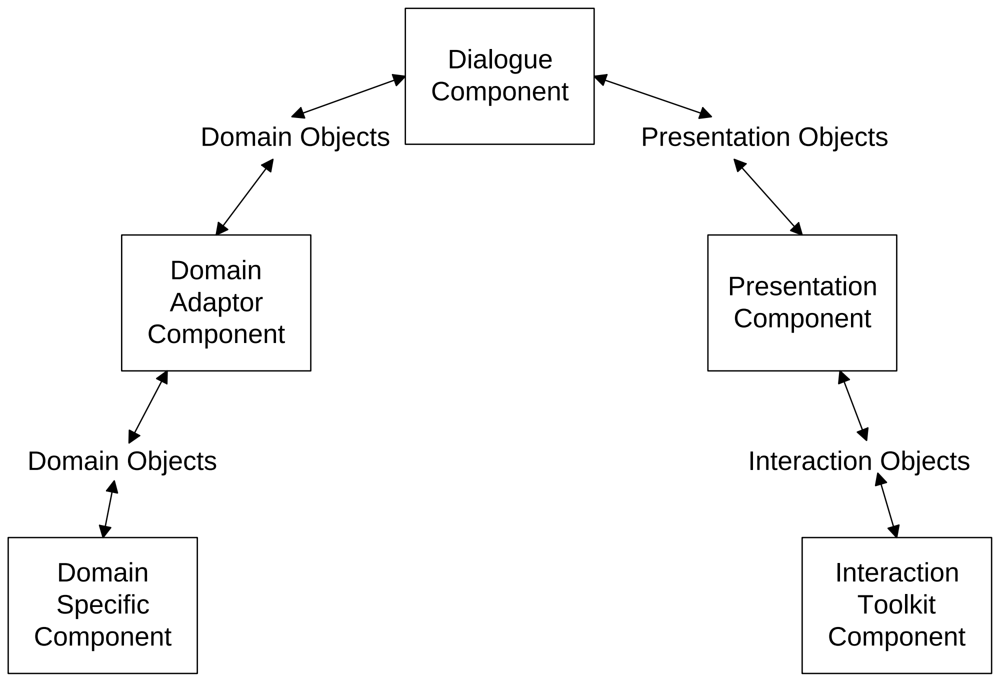


Figure 5: The Arch model [1].

2.3.3 The Arch model

During a series of UIMS Tool Developers Workshops in 1990 and 1991, the participants began to analyse the data exchange and the internal functions of interactive systems. This led to the definition of a runtime architecture, the Arch model [1]. The Arch model is presented in Figure 5.

The application (domain-specific component) and the UI toolkits form the two bases of the Arch model. Other essential functionality is provided by the three additional components, the dialogue, presentation and domain-adaptor components. The *domain-specific component* controls, manipulates and retrieves application data and performs other application-related functions. The *interaction toolkit component* implements the physical interaction with the end user [1].

The *dialogue component* has the responsibility for mapping backward and forward between application-specific formalisms and user interface specific formalisms. The *presentation component* provides a set of toolkit-independent objects for use by the dialogue component. Decisions about the representation of media objects are made in this component. The *domain-adaptor component* triggers application-initiated dialogue tasks, reorganizes domain data, and detects and reports semantic errors [1].

Figure 5 shows not only the components of the Arch model but also the types of objects that cross the boundaries between them. In the Domain-Adaptor Component the *domain objects* and operations are used to implement operations on domain data that are associated with the user interface. The *presentation objects* are virtual interaction objects that include data to be presented to the user and events to be generated by the user. The *interaction objects* are specially designed instances of media-specific methods for interacting with the user [1].

The participants of the Tool Developers Workshop [1] concluded that no single architecture would satisfy all of the possible goals that developers can have in designing interactive systems. Therefore, a metamodel called Slinky was introduced. In the Slinky model the Arch model is generalized to emphasize chosen design criteria. The functionalities can be shifted from component to component in the architecture depending on the goals of the developers, their weighting of development criteria, and the type of system to be implemented. For example, if system performance is the overriding criterion, one puts less emphasis on Dialogue and Domain-Specific Components. Rather, data values may be passed more directly from the Domain-Specific Component to the Presentation Component [1].

2.4 Agent-based models

An overall decomposition, like carried out in the Arch model, is not always sufficient for designing a particular software architecture. In such cases, agent-based models are an useful alternative. They structure an interactive system as a collection of specialised computational units called agents. An agent has a state and is capable of initiating and reacting to events [8].

Agent-based models support especially modularity, parallelism and distributed computing, and suite, therefore, very well for the iterative design of user interfaces for distributed applications and multi-threaded dialogues. Agent-based models push forward the functional separation of concerns introduced by the Seeheim model. Furthermore, the agent-based models generalise the distinction between the concepts and their presentation by applying the separation at every level of abstraction. Some problems concerning agent-based models are that they increase system complexity and may decrease efficiency ([8], [24]).

In this section, two agent-based model, MVC and PAC are introduced. PAC-Amodeus is hybrid model based on PAC and Arch-model introduced in the previous section.

2.4.1 MVC - Model View Controller

The Model View Controller (MVC) paradigm was first designed for user interfaces in applications implemented with the programming language *Smalltalk*, but has since became a frequently used design paradigm for user interfaces, independently of the used language. The MVC model, illustrated in Figure 6, divides an interactive system into three components, each specialized for its task. The *model* contains the application data and manages the core functionality. The *view* manages the visual display of the model and the feedback to the user. The *controller* interprets the mouse and keyboard inputs from the user, commanding the model and the view to change appropriately [5].

The model, view and controller involved in the MVC triad must communicate with each other to enable a proper communication of the application and the end user. The model may be *passive*, which means that it is totally unaware of the existence of either the view or the controller. This is the case, for example, if the model is a text that can only be changed by the user. In many cases, however, the model must have a link to the view to be able to inform the view on changes made to its state by internal domain procedures. The view and the controller are always connected to each other. The controller communicates with the view to determine which objects are being manipulated by the user and calls models methods to make changes on these objects. The model makes the changes and notifies the view to update [5].

The application view usually includes several nested MVC views. The controllers

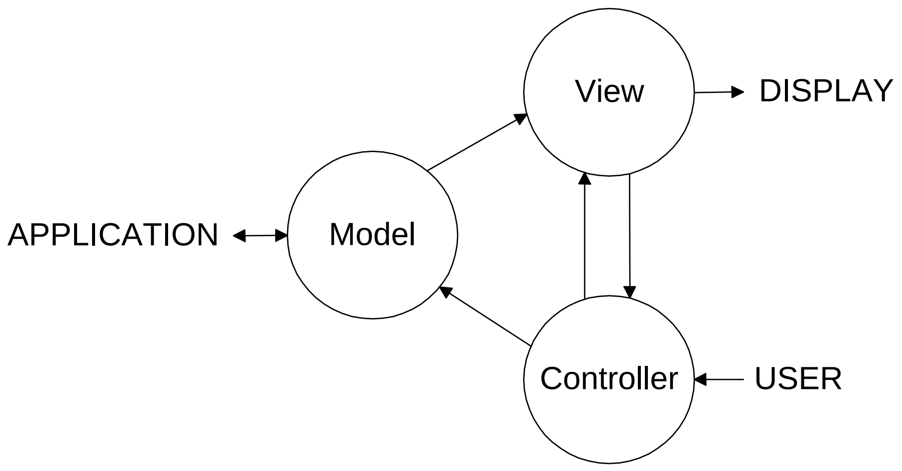


Figure 6: MVC model for interactive systems [5].

of these views must cooperate to ensure that the proper controller is interpreting the user input. For this purpose they form a hierarchical tree, where messages pass from controller to controller along the branches of this tree. Only the controller that has the focus takes an action [5].

Each view is associated with a unique controller and vice versa, but a model can have more than one view-controller -pairs at a time. Any time the model is changed, each dependent view must be notified so that they change accordingly. The possibility to have multiple, even synchronized, views, is a significant benefit of the MVC model. The liabilities include increased complexity, inefficient data access in the view and difficulty to use the model together with user interface toolkits ([5], [24]).

2.4.2 PAC and PAC-Amodeus

In PAC (first introduced in [7]) the interactive system is modelled as a set of PAC agents forming a hierarchy (see Figure 7). P, A and C (Presentation, Abstraction and Control) are the facets of an agent that express different but complementary and strongly coupled perspectives of the same entity. The arrows illustrate the information flow between the agents. There is also an information flow between the facets of an agent [8].

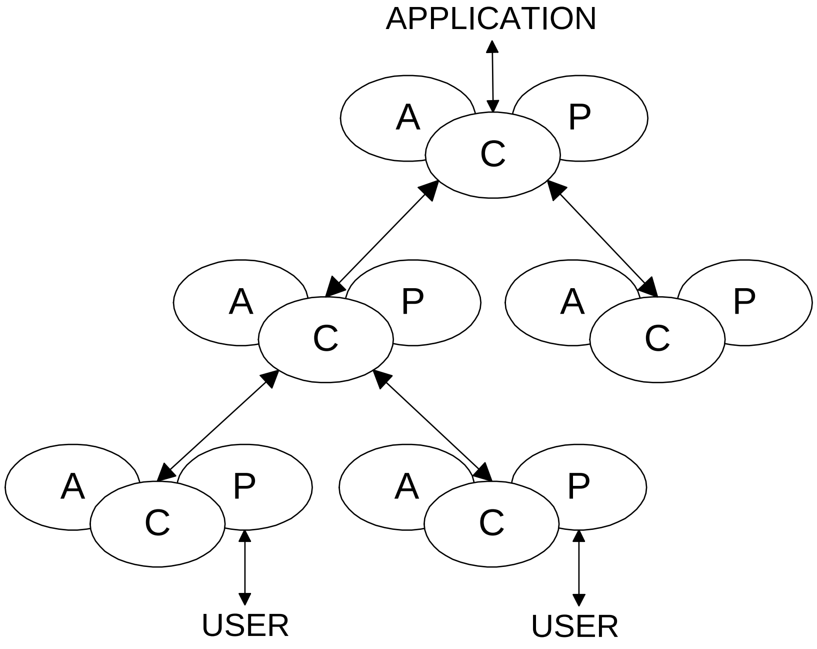


Figure 7: A PAC agent-based model [8].

The *Presentation facet* is responsible for the input and output behaviour and the *Abstraction facet* is the functional core of the agent. The *Control facet* is in charge of communicating with other agents as well as expressing dependencies between the Abstract and Presentation facets of the agent. No agent Abstraction is authorized to communicate directly with its corresponding Presentation and vice versa

[8].

The hierarchy of agents has one top agent and several intermediate and bottom level agents. The *top-level agent* provides a media-independent representation of the global data model and the functional core of the system. Every *Bottom level agent* represents a concept. Its abstraction facet maintains data essential for the concept and the presentation facet gives a view and user access to this concept [24].

The *intermediate agents* represent combinations and relations between the lower-level agents. They maintain consistency between the agents and take care of the information flow between the levels. Dependencies among agents are transitive: each agent depends on all higher-level agents [24].

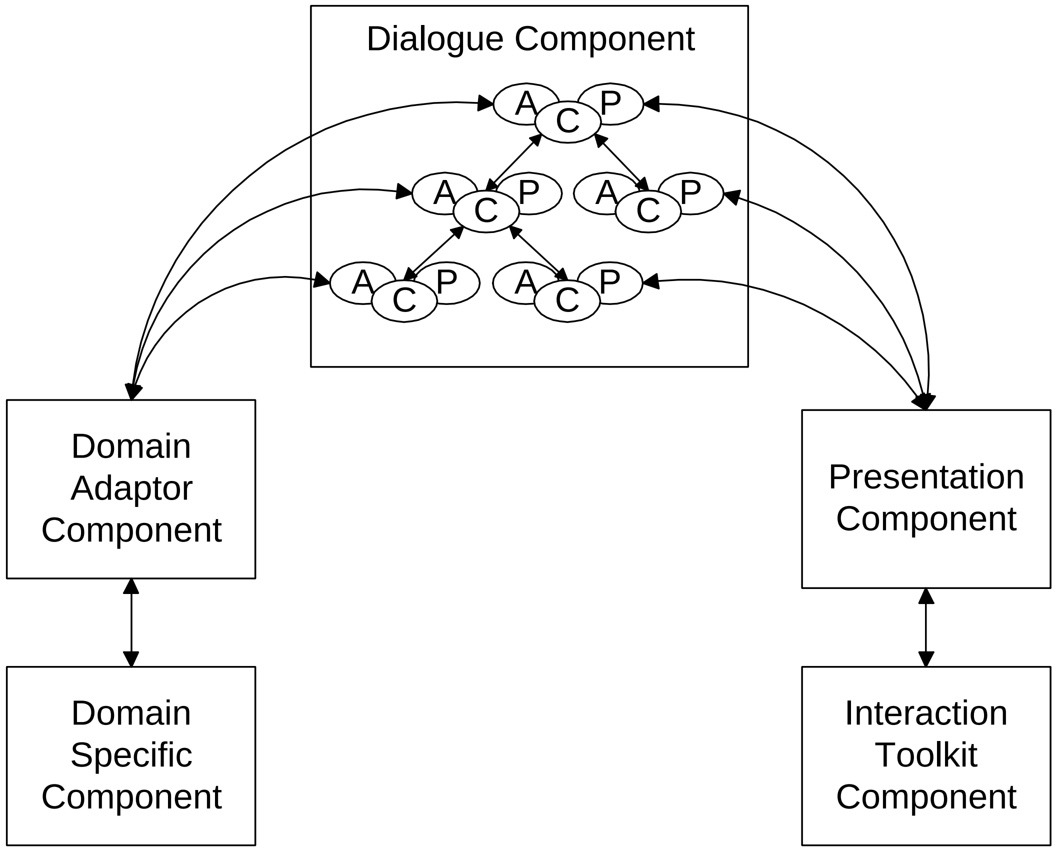


Figure 8: A PAC-Amodeus hybrid model [8].

The PAC-Amodeus model ([8]) is a hybrid model, combining the best of the Arch and PAC models. It uses Arch model as the foundation for the functional partitioning. Arch model supports reusable code and portability, but does not give guidance about how to structure the Dialogue Component. This part is realized using a PAC agent hierarchy. Design pattern introduced in the next section can be used to help the designers to identify the needed agents. The resulting architecture is illustrated in Figure 8.

In contrast to the original PAC style, in PAC-Amodeus the Domain Adapter and Presentation components have direct links to the abstract and presentation facets of the agents in the Dialogue Component. The reason for this design solution is performance. Abstract information from the Domain Adapter may not need additional processing from the top agent or other parent agents, whereas going through the whole PAC hierarchy would be time consuming [8].

2.5 About design patterns

Experienced software designers do not solve every problem from the scratch, but rather reuse solutions that have worked for them in the past. A *design pattern* describes a specific design problem that occurs over and over again in the software development. Then it introduces the core of the solution to that problem. A designer who is familiar with such patterns can use them for design problems without having to discover the solutions again [18].

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as a design pattern makes them more accessible to developers. Design patterns help designers in choosing design alternatives that make the software reusable [18].

Each design pattern systematically names, explains, and evaluates an important and recurring design. The four essential elements of a pattern are pattern name, problem, solution and consequences. Having *names* for patterns help designers talk about them with colleagues and lets designers design in a higher level of abstraction. The *problem* describes when, in which context, to apply the pattern. Sometimes the problem includes a list of conditions that must be met before using the pattern [18].

The *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations. It does not give a concrete design or implementation, but instead provides an abstract description of how general arrangement of elements solves a design problem [18].

The *consequences* are the results and trade-offs of using the pattern. They are important, when the designer is evaluating design alternatives or considering the costs and benefits of applying the pattern. Consequences of a pattern may concern space and time trade-offs, implementation issues or patterns impact on system's extensibility or portability [18].

Gamma et al. [18] introduce a catalogue of tried and tested design patterns. The listed design patterns can help software developers to make design decisions like identifying appropriate objects for the system, determining object granularity or specifying object interfaces and implementations. For example, in the MVC model for interactive systems (see subsection 2.4.1), the managing of information flow from the model to its views can be designed using an *observer* design pattern. In the same way there are design patterns for managing the nested MVC views (a *composite* design pattern) or the view-controller relationship (a *strategy* design pattern). These design patterns can naturally be used also for other user interface architectures [18].

3 Separation of Application and User Interface

Early approaches to interactive system development typically caused user interface and application to be tightly bound together. This led to ever increasing difficulties to make changes on the user interface as the design progressed. It was also hard to support human factors like uniformity of the user interface [20].

One of the main goals of the user interface architectures introduced in the previous section is to separate the application and the user interface. Separation means that the design and implementation of those parts are separated so that changes in one part do not cause changes in the other, as long as both remain consistent with their common internal interface representation. The system should be composed of an user interface, through which all communication between the end user and the system takes place, and an application component, with which the end user does not directly interact [20].

The arguments for the separation include:

- changes to the implementation of the user interface do not cause changes to the implementation of the application, and vice versa
  - impact of change is reduced
  - cost of maintenance is reduced
- the application becomes portable, meaning that it can be executed on more than one platforms with little or no changes.
- the components of the software system become reusable
- it is possible to implement multiple user interfaces for one application
- customisation of the application becomes easier
- the cooperation of user interface designers and application developers is easier
- fast and easy modification of the user interface code and layout is possible
- the program code becomes more readable

- different programming languages can be used for the implementations of the application and the user interface; the application interface provides a boundary where program segments written in different languages meet
- the lifetime of the application gets longer because of the increase in flexibility

The potential decrease in performance is one problem with the separation of the user interface and the application, especially because of increased internal communication among run-time components. Furthermore, for multi-thread, direct manipulation user interfaces, separation into components can be difficult to achieve. In direct manipulation user interfaces there is a need for a closeness of the interface to application semantics (e.g., for semantic feedback in the interface) that works against the separation of user interface and computation (see section 3.2) [20].

The next section 3.1 first considers the questions: what is an application interface and what is an object-oriented application interface. After that, in section 3.2, the connecting of application data to the user interface is discussed.

3.1 The Interface between the User Interface and the Application

Modules and objects of an application have often too much knowledge about their environment. For example, modules or objects know about their user interface details on how their data structures will be displayed or how the user will interact with the application. This is among other things a significant barrier to reuse [2].

There are many ways how a data structure can be displayed, and since this is not an essential property, it should not be attached to the data structure. Similarly, there are many ways how a user can interact with an application and so the application should not be aware of the mode of interaction. An interface that separates the module or object from the user interaction or from the services supplied by another module or object is required. The interface should be aware of the module but the module should not be aware of the interface [2].

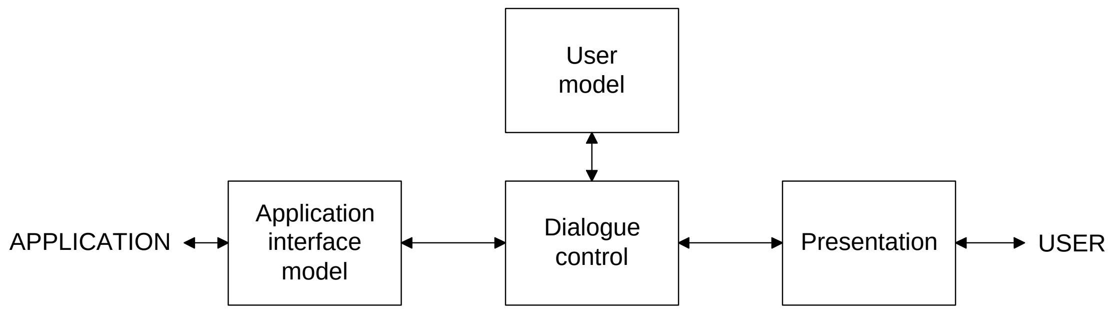


Figure 9: The user interface model from Alty and McKell [4].

The most important part of any software system concerning the separation of the user interface and the application is the application interface placed between them.

Easy modification of the user interface is obtained by a formal definition of the communication between the user interface and application: the application interface. In practice this means that the appearance of the interface to the end user and choices of interaction styles used to communicate with the end user are not known by the application [20].

Alty and McKell [4] propose an interesting model for the application interface, as illustrated in Figure 9. They say that the function of the application interface is to provide an *expert view* of the application. It contains a description of objects and processes that make up the application in expert terms. This expert view provides a solid foundation on which to build a supportive and adaptable user interface. Such higher-level descriptions of the application are more difficult to specify and depend heavily on the user. These extensions to the expert view will be placed in the user model.

This viewpoint includes the idea that with the help of the application interface, it is possible to remove the user interface and use the application directly by calling the appropriate routines in the application interface. This makes it easy, for example, to test the application using test programs.

3.1.1 Object-oriented interfaces

In object-oriented software development the interface between the application and the user interface is a group of class and class hierarchy interfaces (interfaces of abstract base classes). *Namespaces* can be used to group these interfaces [32]. In this section the concepts of object interface and object hierarchy interface are introduced.

The complete set of requests that can be sent to an object is the *interface* to the object. A *type* is a name used to denote a

particular interface. Interfaces can contain other interfaces as subsets: a type is a *subtype* of another if its interface contains the interface of its *supertype* [18].

Objects are only known through their interfaces. The interface says nothing about the object's implementation, meaning that two objects having completely different implementations can have identical interfaces. Software developers can write programs that expect an object with a particular interface, the association of a request to an object will be done run-time, using *dynamic binding* [18].

The main purpose of an *abstract class* is to define a common interface for its subclasses. An abstract class has *abstract operations* that it declares, but does not implement. The implementation is done by the subclasses. Furthermore, a subclass can redefine the behaviour of its parent class by *overriding* its operations [18].

It is possible to define class families using inheritance from abstract classes, where all subclasses can respond to the requests in the interface of the abstract class. Two benefits gained from such a structure are that the applications remain unaware of the specific types of objects they use and, furthermore, the applications remain unaware of the classes that implement these objects. They only know the interface defined by the abstract class [18].

As the application is started up, the used objects have to be created using concrete subclasses. One possibility to do that is to use the so-called *factory methods*, whose only purpose is to create an object of a related subclass. Factory methods, however, expose implementation details and class tree structures by explicitly naming concrete subclasses [31].

A better solution is to create objects using *late creation* that makes it possible to create an instance of a class by only naming the abstract base class and giving a specification for the needed object. This way only methods provided by the abstract base class of the class hierarchy are used and the class hierarchy is encapsulated [31].

The specification can, for example, be a list of properties or an id. Using the specification the base class decides which of its subclasses is referred and creates it. To be able to do this, the base class needs a table of all its subclasses and their specifications. This table can be build in advance using *class retrieval* formalism (see, e.g., [31]).

Using late creation the developers can focus on the relevant abstraction and specification of needed functionality without having to deal with the implementation and naming of the subclasses. Furthermore, it is easier to change the encapsulated class hierarchy, because its class names and structure are hidden [31].

3.2 Connecting application data to the user interface

An early approach to separation of user interface and application was to maintain a strict division of responsibility between them: the application did the work and the user interface communicated with the user [22].

To promote reuse, it is necessary to minimize the amount of information that the user interface has about application internals. On the other hand, researchers generally agree that high-quality user interfaces do not hide the application semantics from the user. For example, a good user interface protects the user from invoking an operation that cannot be executed successfully, or provides default values for commands ([22], [10]).

Several models have been developed to answer the question how much knowledge of application semantics is necessary on the user interface, and how it should be captured (see, e.g., Hurley and Sibert [22]). In this section some possibilities for connecting context and change knowledge and graphics to the user interface are introduced.

3.2.1 Semantic feedback

The rapid information that the system gives to the user concerning the application semantics, the objects the user manipulates, is called *semantic feedback*. It is frequently used in interactive systems using, for example, direct manipulation or natural language as an interaction style [14].

Examples of semantic feedback are error checking and recovery, validation of user input, generation of default values and help for the user. The problem with semantic feedback is that it has both application and user interface aspects and crosses the application interface boundary. Therefore, it may be difficult to modify or extend functionality related to semantic feedback [14].

There are two main classes of solutions for coping with semantic feedback in a software system based on the separation of application and user interface. In the *semantic delegation* approach the responsibility of semantic feedback is put in the user interface part of the system. The other approach is to make the application part responsible for the semantic feedback, resulting a lot of communication between the user interface and the application. A trade-off has to be done between the amount of separation and the amount of communication between the user interface and the application [14].

3.2.2 Contents and change knowledge

Szekely [33] compares some implementation techniques for providing contents and change knowledge for the user interface in terms of their ability to implement a wide variety of programs and user interfaces (*generality*), and in terms of their ability to hide the implementation of user interface design decisions from the implementation of the program's functionality (*modularity*).

The interaction controls need two kinds of knowledge about the application data in order to present it: contents and change knowledge. The techniques for supplying *contents knowledge* can be divided into two classes: external and internal. Interaction controls of the *external* type access contents knowledge by calling routines provided by the application interface. The implementation of concepts and interaction controls are hidden from each other. Interaction controls of the *internal* type access contents knowledge by accessing directly the implementation of the concept types [33].

Due to Szekely [33], external implementations are not as general as the internal ones, because the application interface may not provide enough routines to access all the contents knowledge needed to provide a particular presentation, but from the point of view of modularity, external implementations are better than internal ones, due to benefits mentioned in the beginning of this section.

The techniques for supplying *changes knowledge* to the implementation of interaction controls can also be divided into two classes: announcements and recognition. For *announcements* the routines in the application interface inform interaction controls about a change by calling a routine. A *direct* announcement is a call to the particular interaction control routine that will update the interaction control to reflect the change. *Indirect* announcements are essentially broadcasts about changes that occur in concepts, and any interaction control interested in them can receive them and update their representation appropriately [33].

When *recognition* is used for supplying changes knowledge, the routines in the application interface need not do anything to communicate changes to interaction controls. The interaction controls must monitor the behaviour of their models and recognize when a change of interest to them occurs. The monitoring can be done by monitoring the state of values, or by monitoring the routines that modify the values [33].

An *active value* is a piece of program state with an associated routine that is invoked automatically, if the state is changed. Active values can be used to provide change information for the interaction controls [33].

A *daemon* is a routine that can be associated with another routine so that if the second routine is invoked, then the daemon is also invoked. A *before* daemon is called before the monitored routine is called, and an *after* daemon is called after the monitored routine exists [33].

Techniques based on announcements are more general, because any change can be announced by calling an announcement routine. One weakness of active values is that they can only recognize changes in data structures, and, hence, are unable to recognize changes like invocation or termination of routines. Using active values, it is also difficult to recognize the change at the right level of abstraction. A single change at the conceptual level may require many changes to the data structures. Daemons, on the other hand, cannot recognize changes that occur during the execution of the routine they are monitoring [33].

The biggest problem with announcements is that the program developer has to find the appropriate places in the functionality implementation from where the announcements should be made. Changing a displayed object, but forgetting to call the relevant update routine, leads to errors. Also changing the implementation of the functionality requires making sure that the proper change announcements are still made. The fact that changing the implementation of an interaction control leads to changes of the functionality reduces modularity [33].

3.2.3 Graphical representation

One question concerning the interface is which component, user interface or application, should be responsible for graphical output. In particular, which component is responsible for the mapping of application data structures to the graphical representation?

Enderles [15] view is that this depends on the type of graphics. The application part should be responsible for creating *direct graphics*, which are direct representations of the application database that are created and modified by an interactive process, like film animations. In this case, the mapping of application data structures to a graphical representation should be a task of the application. The user interface, instead, generates graphical output for *symbolic graphics*, like icons, and takes care of their mapping. In both cases, the management of the output is done by the user interface.

For direct graphics, the application data structures often have to be made available for the user interface so that it can control output generation and picking. Duplication of data structures should be avoided to avoid consistency problems. For symbolic graphics, both the interpretations of the pick input and the setting of the picking context are done by the user interface [15].

4 Interaction controls

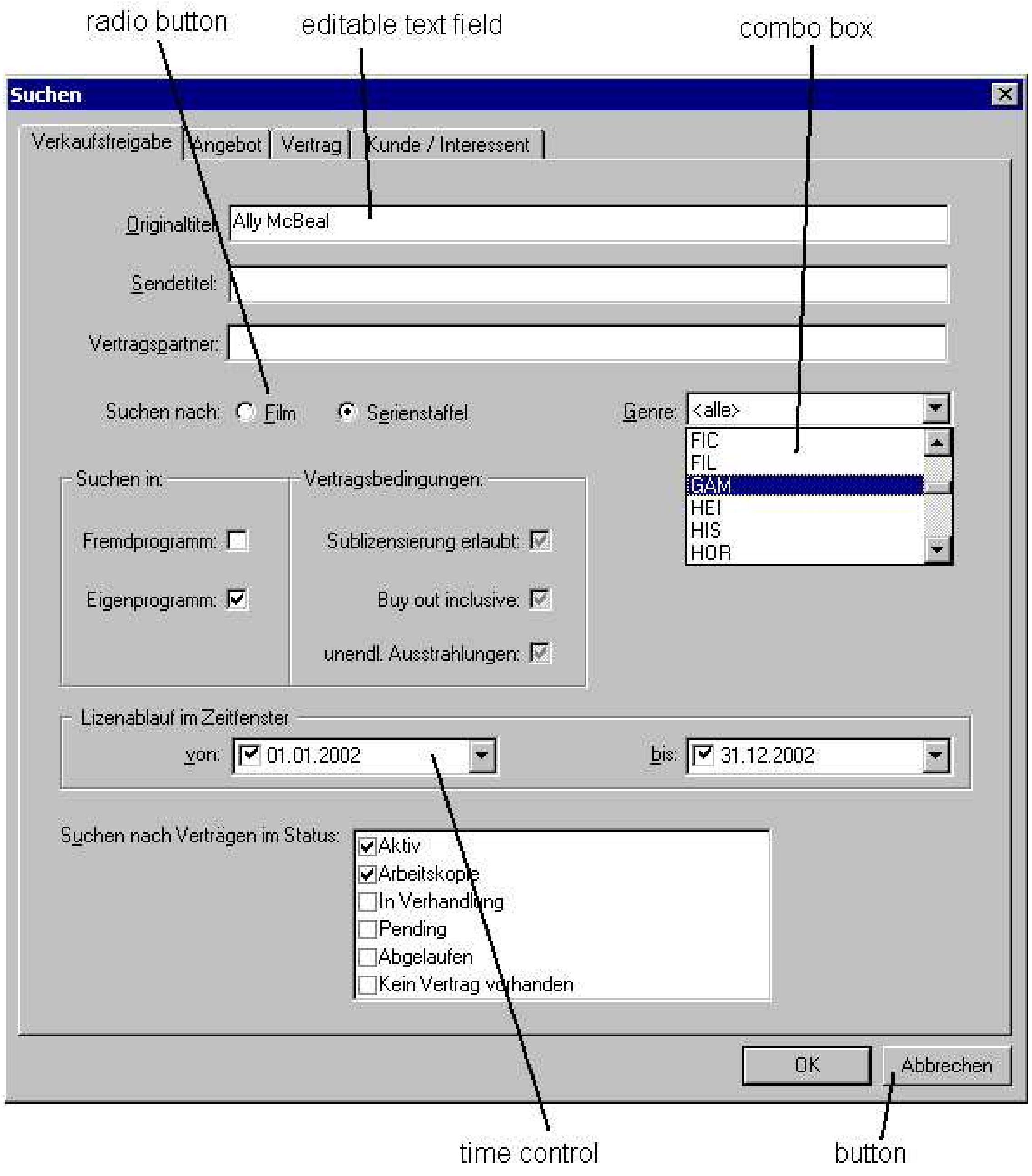
User interface software tools help developers design and implement the user interface. Virtually all applications today are built using toolkits and interface builders. These tools have had time to achieve a high level of sophistication, because the user interfaces have stayed similar for a long time. A small set of constructs invented 15 or more years ago have been widely adopted with only small variations [28].

User interface toolkits are developed on top of the abstractions provided by window managers. They typically provide both a library of interaction components and an architectural framework to manage the operation of interfaces made up of those components [28].

Toolkits help achieving the goal of maintaining interface consistency. The look and feel of an user interface is determined largely by the collection of interaction controls provided for it. Toolkits help to ensure a consistent look and feel among application programs and by using them, considerable programmer productivity can be gained ([28], [17]).

Most user interface toolkits make use of *widgets*, traditional interaction controls like buttons, editable text fields, sliders, combo boxes etc. (see Figure 10). Designers construct user interfaces by choosing and laying out widgets and then connecting them to application semantics. This approach has some problems: most widgets are too low-level and constructing interfaces from them is too much work. Working with these widgets focuses attention on appearance and layout issues, rather than on more important semantic issues. Designers can easily make poor widget choices, yielding poor interfaces. In addition, the widgets know nothing about the variables they control, and, therefore, they do not deal with application semantics properly [23].

The widgets clearly belong to the presentation part of an user interface architecture. In this chapter some techniques are introduced that go beyond the user interface widgets and provide methods for higher level of abstraction, which support the separation of user interface and application and link semantic information to the interaction controls.



Originaltitel: Ally McBeal

Sendetitel:

Vertragspartner:

Suchen nach: Film [✓ Serienstaffel]

Genre: <alle>

FIC
FIL
GAM
HEI
HIS
HOR

Suchen in:

Fremdprogramm: ☐

Eigenprogramm: ✓

Vertragsbedingungen:

Sublizensierung erlaubt: ✓

Buy out inclusive: ✓

unendl. Ausstrahlungen: ✓

Lizenablauf im Zeitfenster

von: ✓ 01.01.2002 bis: ✓ 31.12.2002

Suchen nach Verträgen im Status:

✓ Aktiv
✓ Arbeitskopie
☐ In Verhandlung
☐ Pending
☐ Abgelaufen
☐ Kein Vertrag vorhanden

Figure 10: A typical view on the user interface in S|4|Rights.

When the functionality and semantic information of an interaction control is separated from its presentation, the user interface look is easier to change without changing the functionality of the control.

4.1 Model-based and automatic techniques

The goal of the investigation of automatic techniques for generating user interfaces is generally to allow the designer to specify interfaces at a very high level, with the details of the implementation to be provided by the system. It is hoped that programmers without user interface design experience could just implement the functionality and rely on these systems to create high-quality interfaces. Further, there is the promise of benefits such as automatic portability across multiple types of devices [28].

Model-based systems use techniques like heuristic rules to automatically select interaction controls, layouts, and other details of the interface. Automatic and model-based techniques suffer from unpredictability: the connection between specification and final result can be quite difficult to understand and control. This is one of the reasons why this techniques have not found wide acceptance. The need for model-based or related techniques, however, raises as new requirements for device independence emerge [28].

The XML User Interface Markup Language (UIML) is a technique used to define the location and design of controls. It also defines actions to take when certain events take place. UIML allows designers to describe the user interface in generic terms, and then use a style description to map the interface to various operating systems. The universality of UIML makes it possible to describe a rich set of interfaces and reduces the work moving the user interface to another platform [27].

The idea of defining properties of interaction controls that enable grouping or evaluating them brings the controls on a higher level of abstraction. In the next sections two models for interaction controls are introduced that define some properties of controls, but instead of automatically selecting the presentation, they give the designer some alternatives to choose from.

4.2 Selectors

Johnson et al [23] have developed an application construction environment (ACE), where developers work with semantic-based interaction controls called *selectors*. The selectors are classified according to their interface semantics rather than their appearance. Each type of selector can be presented in a variety of different ways that can be chosen semi-automatically. The selectors are linked to the application semantics: their values are application data-types and can be used directly by the application. ACE's approach is to focus concern on specifying the role of the interaction components in the user interface and the connection between the user interface and the application semantics.

The primary components of ACE applications are not widgets but *visual formalisms* (VF), which are reusable components that embody significant pieces of functionality commonly found in interactive systems and their familiar presentations. Tables and graphs are examples of VFs. However, simple interaction controls are needed for controlling aspects of VFs. The designer selects interaction controls based on the semantics of the various choices and settings that the application has. After that they can choose from some presentations that the system has to offer for the specific interaction control. The idea is that in this way the designers pay more attention to application semantics [23].

The selectors have a base class, also called *Selector*. Data Selectors and Command Selectors are subclasses of the class Selector (see Figure 11). *Data Selectors* are selectors that display and set application variables, having restrictions on the values they may assume. First, they are restricted to a particular static data type or class, called the *base-type* of the Data Selector. Additionally, Data Selectors are restricted to those values of the base-type that are contained in a specific set, referred to as the *domain* of the Data Selector. For example, a Data Selector could have a base-type Colors and a domain {red, green, blue}. Finally, the Data Selectors value is a set, having a specific minimum and maximum. For example, one Data Selector may offer a choice from zero to five selections, while another Data Selector with the same domain may allow only

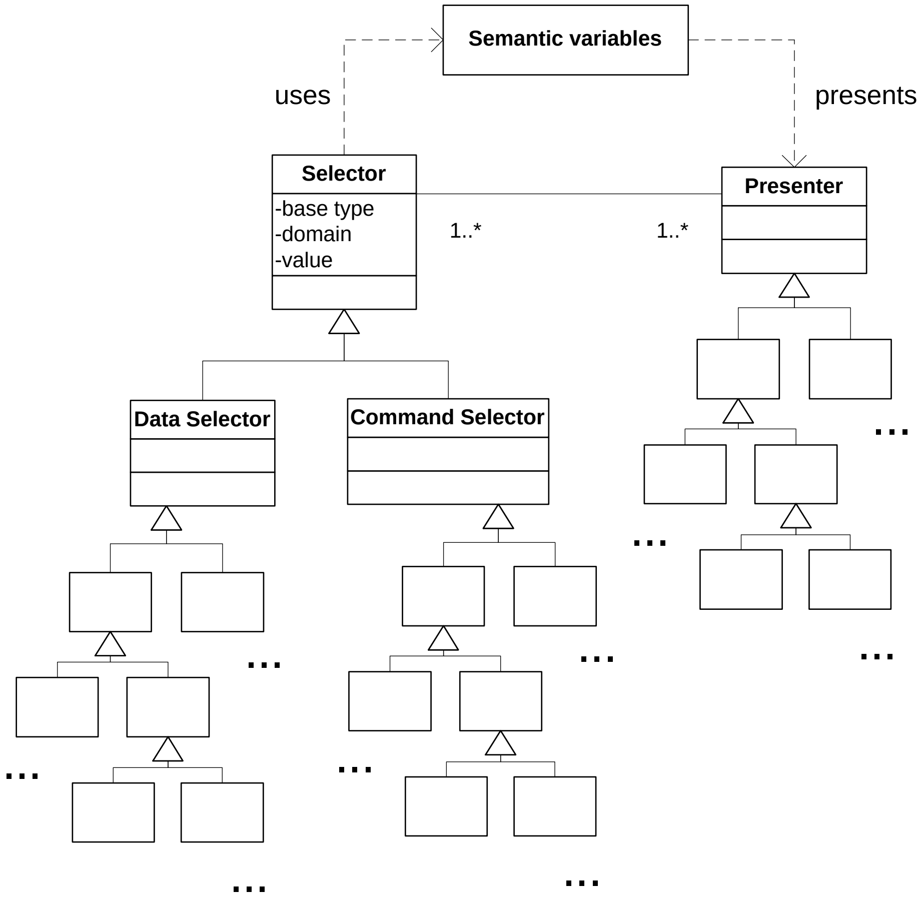


Figure 11: The structure of selectors and presenters [23].

one selection [23].

Data Selector *Presenters* encapsulate the presentation and editing of choices. They are based on the assumption that there is a protocol between semantic variables (e.g. numbers, colors, times) and the interfaces that allow users to set them, and that alternative Presenters can be selected based on features (e.g. textual vs. graphical, tall vs. wide). This protocol between Data Selectors, Presenters, and basic semantic variables allows Data Selector Presenters to concentrate only on the overall appearance and behaviour of the settings, leaving the display of individual values (e.g. current value, the possible values) to a Presenter for the semantic variable [23].

The triad selector, presenter and the semantic variable presented can be compared to the triad controller, view and model in the MVC architecture presented in subsection 2.4.1. The difference is that a selector is not bound to one specific presenter but the designer can choose from different presenters.

By crossing the semantic variables with the presentations, ACE provides a much greater variety of interaction controls than most UI toolkits, increasing the likelihood that the right presentation for a particular application setting can be constructed.

In ACE, the applications provide *operations*, which allow users to manipulate data. Operations take arguments, have preconditions, and produce effects on application data-state. Operations are invoked by *commands* that gather arguments for operations, test preconditions, and request confirmation if needed. The logic of *Command Selectors* is similar to that of Data Selectors: Command Selectors domain is a set of objects of the type command and several Command Selector Presenters are available for Command Selectors [23].

4.3 A semantic model of interaction controls

Zhou and Houck [34] have developed a model of interaction controls similar to that of selectors. They do not, however, express how the application data should be connected to the interaction control. They rather supply a model for describing interaction controls and for designing controls and control families.

Zhou and Houck [34] model the semantics of an interaction control from three aspects: intentional, presentational and behavioural. In other words, they describe a control by answering the questions: why an interactions control is created, how it appears as presentation element, and how it behaves when acted on. To be precise, an interaction control is according to Zhou and Houck a six-tuple:

```
C = <Intent, Host, Style, Form, Operation, State>
```

Intent specifies the type of user interaction tasks for which an interaction control is designed. For example, a control may be created for navigation or analysis (e.g. comparing, classifying). The intent of a control is described at multiple levels of abstraction [34].

The term *Host* is used to refer to a visual object that contains the control. The *Style* of a control may be *embedded* or *attached*. Embedded controls do not have their own visual form, but are integrated, for example, in an icon. Attached controls, on the other hand, have visual forms that directly encode the interaction content (e.g. slider). The *Form* characterizes the visual description of a control. The host, style and form together define how an interaction control appears as part of the visual presentation [34].

Operation specifies how a control responds to interaction activities. Each operation is expressed by a four-tuple

```
Op = <Trigger, Pre, Act, Pos>,
```

where a *Trigger* is a set of interaction activities that can invoke a response, *Act* is the action taken, *Pre* describes the preconditions that must be true before executing the action and *Pos* represents the post conditions that become true after the execution [34].

The *State* of a control captures its *visibility* and *availability*. Therefore, the State describes the dynamic status of an interaction control caused by control operations.

An example of an interaction control definition is given in Figure 12. The `profileCtrl` is used in an interactive system where the user can explore residential properties. Using the control the user obtains the profile of a house. This control is embedded in a house icon.

Zhou and Houck [34] describe all interaction activities, actions, and conditions using abstract methods. Usually all controls share a set of standard actions, like create, show or enable. Other actions are added for each particular control.

Using this semantic model, Zhou and Houck [34] classify controls into families based on their intent. This way, every control is a member of a control family (class), and all control families share a set of common features (e.g. Host and Style) and standard operations. The control classes are organized into a control hierarchy according to their level of abstraction. Particularly, Zhou and Houck [34] identify three control

```
profileCtrl <
  Intent: PROFILE(House)
  Host: House
  Form: NULL
  Style: EMBEDDED
  Operation: {
    Trigger: Toggle(self)
    Precondition: available(self) & invisible(House.Profile)
    Action: Reveal(House.Profile)
    Postcondition: revealed(House.Profile)}
>
```

Figure 12: A semantic definition of an interaction control.

classes: navigation controls, data analysis controls and visual manipulation controls. The control hierarchy serves as a base for building new controls systematically.

5 The user interface architecture and interaction controls in S|4|Rights

In this chapter, the user interface architecture and application interface in S|4|Rights are first introduced. After that the focus is set on one part of the user interface architecture, the interaction controls and their hierarchy in S|4|Rights.

In section 5.3 the way how application data is connected to the user interface in the user interface architecture of S|4|Rights is introduced. Some extensions made to the interaction control hierarchy in S|4|Rights are handled in section 5.4. These examples demonstrate some advantages gained from the user interface architecture, for example, how easily a new property for all controls can be added in the interaction control hierarchy.

After that a different kind of interaction control is introduced: a navigation tree. Instead of presenting a value of one column in a database table, the navigation tree in S|4|Rights illustrates information from several database tables at the same time. Here the problem of connecting the application data to the user interface without breaking the separation of the user interface and the application becomes essential. First the problems due to the old navigation tree are described. Then a new navigation control that connects the application data to the user interface in a different way is introduced.

5.1 The user interface architecture in S|4|Rights

The basis of the user interface architecture in S|4|Rights is determined by the MFC framework (see subsection 1.1.1). Figure 1 shows that the user interface part of the system consists of a view inside a frame window. This is a very simplified structure and will now be introduced in detail.

Figure 13 shows the hierarchy of the user interface objects in S|4|Rights. The frame window is divided into two views and has in addition a toolbar (menu and button bar) on the top and a status bar on the bottom of the frame window. One of the views contains only an interaction control, namely the navigation control (see section 5.5).

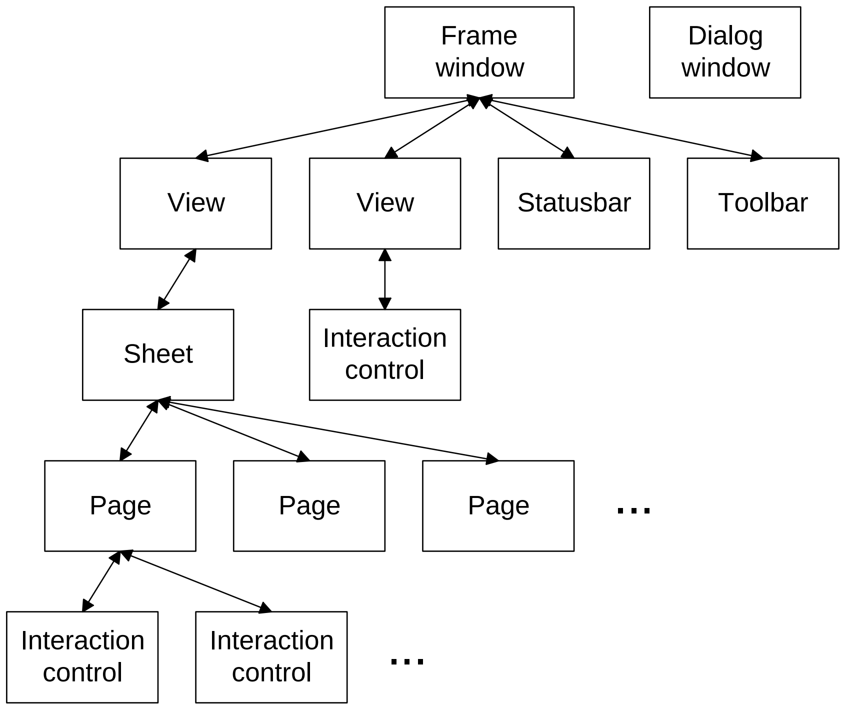


Figure 13: The hierarchy of the user interface objects in S|4|Rights.

The other view contains a sheet, which is a base for several pages. Only one page is visible at a time, but the user can navigate between the pages. Every page has usually several interaction controls, text fields and lines that form groups out of interaction controls. The dialog window -box next to the frame window indicates that such separate windows are sometimes used in the user interface for supporting the frame window or for informing the user. The dialog windows can also contain a sheet and pages or simply text and interaction controls.

The hierarchy of the user interface objects in Figure 13 can be compared to the structure of agents in the PAC model in subsection 2.4.2. Frame window is the top-level agent of the hierarchy. Views, sheets and pages are intermediate agents that build combinations of the lower-level agents. The end user communicates with the bottom level agents: interaction controls, toolbar and status bar. The difference to the PAC model is that in S|4|Rights the objects communicate with each other using abstract base classes as interfaces (see subsection 3.1.1), in PAC model the agent communicate using the control facet instead of the abstraction facet.

Figure 14 shows the user interface architecture in S|4|Rights. It is very similar to the PAC-Amodeus architecture introduced in subsection 2.4.2. The main difference is that the presentation component in PAC-Amodeus model (see subsection 2.3.3), whose purpose was to provide a toolkit-independent abstraction for the dialogue control, is not needed, because of the common abstraction of the object hierarchies through abstract base classes. The names of the components in Figure 14 are based on the Seeheim model in 2.3.1. Note that the presentation in Figure 14 corresponds to the interaction toolkit component in the PAC-Amodeus model, not the presentation component. The dialog window was left out from Figure 14 to keep it simple. Its connections to other user interface objects are similar to those of the frame window.

The application interface in the user interface architecture of S|4|Rights consists of abstract base classes (see subsection 3.1.1) for the user interface objects (this is illustrated in Figure 14 through the lines between the application interface and the single user interface objects). Dialogue control contains the functionality of the objects. The classes used for the presentation of the user interface objects are located in the presentation component.

The interaction control hierarchy introduced in the next section is one part of this user interface architecture. Although at first sight different, the Figures 14 and 15 show the same structure: in Figure 15 the parts of the interaction controls are represented in the corresponding parts of the user interface architecture. `CCtrlBase` is the abstract base class for all interaction controls and belongs therefore in the application interface part of the user interface architecture. The functionality of the interaction controls is in the dialogue control part of the architecture and the classes used for the presentation are in the presentation part of the user interface architecture. For the interaction controls in Figure 15 the presentation part consist of the MFC control classes.

The handling of the interaction controls in sections 5.2, 5.4 and 5.5 will show in practice how well the chosen user interface architecture in S|4|Rights works and how it makes the user interface more structured, easily modified, extendible and separable.

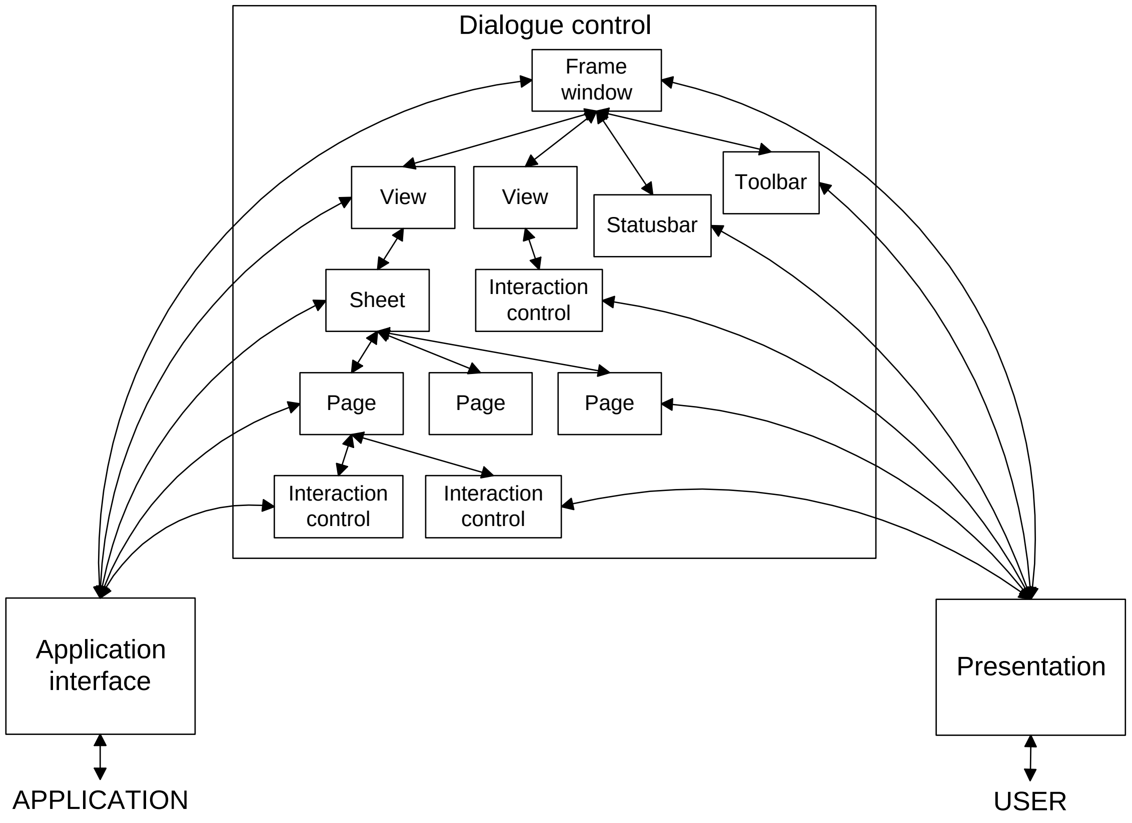


Figure 14: The user interface architecture in S|4|Rights.

5.2 The interaction control hierarchy in S|4|Rights

The models introduced in sections 4.2 and 4.3 form a hierarchical structure, to be exact, a tree of the interaction controls. The root of the tree is an abstract base class, the knots are control families and the leaves are controls. The interaction control hierarchy in S|4|Rights has also an abstract base class for the controls. This hierarchy, with some simplification, is illustrated in Figure 15.

This kind of a structure has many benefits. The abstract base class works as an interface between the controls and the application. Typical operations executed on the interaction controls are common operations for all interaction controls and are triggered through the base class. Therefore, a procedure using them does not have to know, what kind of control (edit field, combo box,..) it is operating and common procedures using a pointer to the base class can be used instead. Furthermore, this yields the advantage that when an interaction control on the user interface is changed to another one, changes on these procedures are not needed.

Every Selector in 4.2 has a Presenter-object, which encapsulates the presentation of the interaction control on the user interface. Johnson et al [23] do not specify how the Selectors and their Presenters are connected to each other. The control hierarchy in S|4|Rights encapsulates the presentation of the interaction controls using multiple inheritance. The control classes, like `CCtrlButton` or `CCtrlEdit`, are inherited from the abstract base class `CCtrlBase`, but each of them has also a second base class providing the user interface presentation and access.

The MFC controls, like `CButton` and `CEdit`, are used for this purpose. The older parts of the system used these interaction controls directly, causing the corresponding code to be bound to a specific interaction control, like an edit field. In this kind of user interface programming, it is hard to make changes to the user interface, like replacing an edit field by a combo box, because these changes mostly concern many parts of the code. Often it can also be necessary to have many functions for one functionality, one for each interaction control that needs the functionality.

Usage of multiple inheritance is an important design solution in the user interface

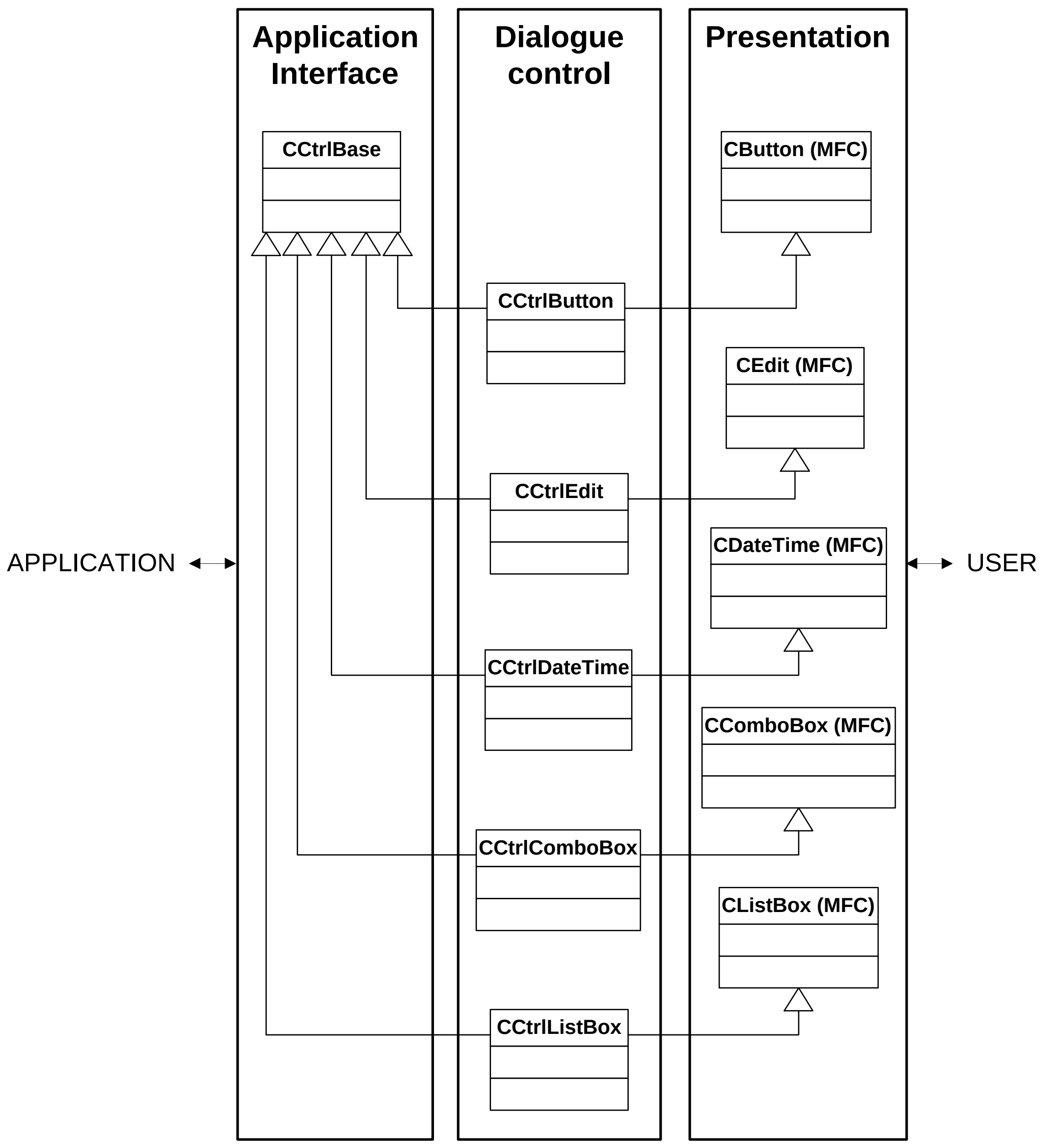


Figure 15: The hierarchy of the interaction controls and division to the levels of the Seeheim model.

architecture of S|4|Rights. Multiple inheritance enables the complete separation of the application interface and the presentation. For example, `CCtrlBase` can only use functionality of `CEdit` through `CCtrlEdit`, and vice versa.

For adding a new interaction control to the hierarchy, it is sufficient to implement a new interaction control class, which is inherited from `CCtrlBase` and overrides the essential methods in `CCtrlBase`. For the appearance on the user interface the new control class must also be inherited from a suitable interaction control class like those in MFC. If one wants to change the presentation of the controls on the user interface, changes are only needed in the dialogue control -component for a wide range.

The layout of the user interface in S|4|Rights is defined in resource files produced with Microsoft Visual C++ resource editors. In these resource files the used interaction controls and details of their appearance on the user interface, like color or font or the scrolling style for edit fields etc., are defined.

5.3 Connecting application data to the user interface in S|4|Rights

One common property of the models for interaction controls in sections 4.2 and 4.3 is that they join the interaction controls to the semantics of the application. The modules in S|4|Rights provide an interface to a database and, therefore, values in the interaction controls on the user interface are values of single fields in a database table (one exception is the navigation control introduced in section 5.5 that displays information from several database tables at the same time). The values are saved in the form of database objects, having the same data type than the database column they represent (string, integer, time, bool, etc.). The database objects have a common base class, `CDbCol`, which is used in common procedures for the interaction controls.

`CDbCol` encapsulates the properties base-type, domain and value of an interaction control introduced in section 4.2 for the selectors. Database column classes inherited from `CDbCol`, like `CDbColString`, `CDbColMoney` and `CDbColTime`, determine the used base-type and accept only valid values.

`CDbCol` works as an interface between the application data and the user interface. The `CDbCol`, together with `CCtrlBase`, provides functionality for contents and change knowledge (see subsection 3.2.2). This means that the user interface architecture in S|4|Rights uses contents knowledge of external type and direct announcements for supplying change knowledge, because these solutions support the separability of the user interface.

The database column classes also provide some semantic feedback functionality (see subsection 3.2.1), for example, default values for interaction controls. The validation of user input is done in `CCtrlBase`, using functionality of `CDbCol`. Further actions are needed only if the entered value in the interaction control is valid.

With the new navigation control an additional connection between the application data and the user interface is defined: `CNavData`. This class is handled in detail in sections 5.5.3 and 5.5.4.

In some rare cases the application interface is gone around, meaning that the user interface architecture of S|4|Rights has a "switch", like the KBFE architecture introduced in subsection 2.3.2 (see Figure 4). For example, some features of interaction controls are sometimes changed in runtime using direct calls of functions in the object interfaces of those specific interaction controls.

5.4 Extensions to the interaction control hierarchy

One important characteristic of a proper architecture design is the robustness to changes. A design that does not take changes into account risks major redesign in the future [18]. In this section some extensions to the control hierarchy in S|4|Rights are introduced. The easiness of making these extensions is one prove for the usefulness of the user interface architecture.

5.4.1 Uniform handling for cut, copy and paste

In the MFC framework the handling of cut, copy and paste events is done in the base class for views, `CBaseView`. First, one procedure is called to verify whether the demanded command is permitted or not, and if the answer is yes, another procedure carries out the action. For example, for cut these procedures are called `OnUpdateCut` and `OnCut`.

When the control hierarchy did not exist and the MFC controls were used directly, the handling had to be done explicitly for every interaction control. As an example, a pseudo code for a part of the handling for cut, is introduced here :

```
CBaseView::OnUpdateCut()
{
    CWnd* poWnd = getActiveControl();
    CButton* poButton = dynamic_cast<CButton*>(poWnd);
    if (poButton != NULL) {   // poWnd is a button
        ...
    }
    CEdit* poEdit = dynamic_cast<CEdit*>(poWnd);
    if (poEdit != NULL) {   // poWnd is an edit control
        ...
    }
    CComboBox* poComboBox = dynamic_cast<CComboBox*>(poWnd);
    if (poComboBox != NULL) {   // poWnd is a combo box
        ...
    }
    ...
}
```

In the `OnCut`-procedure the handling of different interaction controls was implemented similarly. If, for example, a new control was added to the system, all procedures like this had to be changed. Of course, these procedures work also against the encapsulating of interaction control handling behind the application interface.

To remove these problems, new user interface architecture and corresponding control hierarchy was created, where handling of these events was changed so that only procedures of the application interface are used. CCtrlBase was extended with procedures for verifying the permission (canCut, canCopy, canPaste) and for the actions (cut, copy, paste). In the base class, the permission procedures simply return false and the action procedures do nothing. This way, if a specific interaction control does not support the handling of cut, copy and paste, the corresponding procedures do not have to be implemented in that control class. Otherwise, the procedures are overridden in the dialogue control component of the interaction control so that they implement the functionality for the specific control.

After these modifications the handling for cut in the CBaseView-class looks as follows:

```
bool CBaseView::OnUpdateCut()
{
    CWnd* poWnd = getActiveControl();
    CCtrlBase* poCtrl = dynamic_cast<CCtrlBase*>(poWnd);
    if (poCtrl != NULL) {   // poWnd is an CCtrlBase-object
        return poCtrl->canCut();
    }
}

CBaseView::OnCut()
{
    CWnd* poWnd = getActiveControl();
    CCtrlBase* poCtrl = dynamic_cast<CCtrlBase*>(poWnd);
    if (poCtrl != NULL) {   // poWnd is an CCtrlBase-object
        poCtrl->cut();
    }
}
```

References to specific control classes are not needed any more; only the application interface of the user interface architecture, CCtrlBase, is used. If the functionality of cut, copy or paste changes for an interaction control, only the control class in the dialog control component needs to be modified.

5.4.2 undo and redo for the controls

Some functionality for the interaction controls can be implemented solely in the base class in the application interface of the user interface architecture without having to change the implementation of the control classes in the dialog control component. One example for this in S|4|Rights is the undo and redo -functionality, called `CCtrlHistory` (compare with the *Command* design pattern in [18], section 2.7). It does not remember the low level user activities like mouse clicks or entered characters, but saves the accepted values for every interaction

control. The user can then switch between the values in one interaction control.

The base class for interaction controls, `CCtrlBase`, keeps a vector of the accepted values in the control. It is easy to implement this functionality in the base class, because the values in the interaction controls are objects from a common object hierarchy. This means that no matter if the interaction control has a value type int, string or bool, the values can be saved in the vector using pointers of the base class for these control value classes, `CDbCol` (see section 5.3).

The `CCtrlHistory` is valid as long as fields of one record are edited on a page. As soon as the user selects another record and changes a value of it, the `CCtrlHistory` for that specific interaction control will be cleared. This constraint could be removed by saving the control history in the database.

The history mechanism in S|4|Rights could in addition be advanced by adding a page history to the system. The base class for pages, `CPageBase`, could keep a vector of `CCtrlBase`-pointers remembering the order in which the values of the interaction controls have been modified. When the user wants to undo his action on the page level, the undo method for the right interaction control could be called due to the

information saved in the base class for pages.

5.4.3 Interaction control families

As already mentioned in section 5.2, the interaction control models introduced in sections 4.2 and 4.3 group the controls in families based on their semantic features or intent. Although the control hierarchy in S|4|Rights consists of a quite small amount of controls, the need to group them arose from special needs of some controls: `CCtrlComboBox` and `CCtrlListBox` differ from the other interaction controls in such a way that they present many values at the same time. This is why they need some functionality, like adding a new value to the list they present or handling the selection of a specific value in the list that does not make any sense for the other controls like `CCtrlButton` or `CCtrlEdit`. Furthermore, some procedures in the system work only with combo boxes and list boxes.

Using the control structure introduced in subsection 5.2, the programmer has to make a dynamic downcast from the `CCtrlBase`-pointer to a `CCtrlComboBox` or `CCtrlListBox`-pointer to be able to access such a procedure. These downcasts are not optimal, because they go past the application interface and make the code unreusable.

This problem was solved by extending the application interface of the interaction control hierarchy, like illustrated in Figure 16. For `CCtrlComboBox` and `CCtrlListBox` a new base class, `CCtrlEnumBase`, was added. `CCtrlEnumBase` is inherited from `CCtrlBase`. In procedures for combo boxes and list boxes the programmer now makes a dynamic downcast from `CCtrlBase` to `CCtrlEnumBase` without having to pass the application interface. Additionally, on the user interface a combo box can now be changed to a list box without any changes to the business logic, because procedures that earlier worked only with one of these controls, now use a `CCtrlEnumBase`-pointer.

As the control hierarchy in S|4|Rights grows, similar control families can be added to the control hierarchy without making changes to the existing hierarchy. The grouping of the interaction controls extend the changeability of the user interface. Every group could contain, for example, simplified controls that could be used in devices with smaller displays.

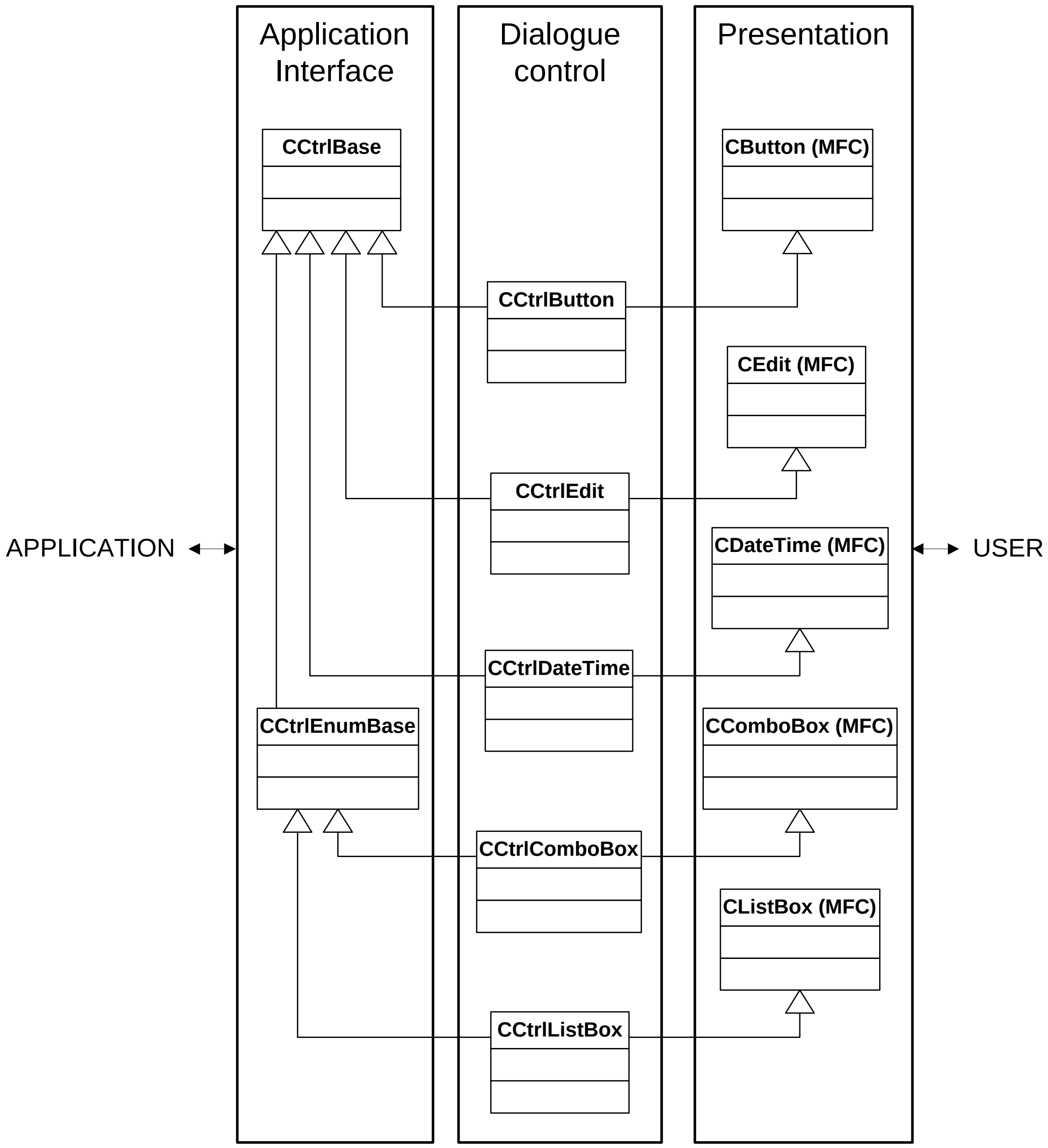


Figure 16: The extended hierarchy of the interaction controls.

5.5 A navigation control

All the modules in S|4|Rights provide an interface to a database. The values in the interaction controls on the user interface are mostly values of single fields in a database table. An exception to this is the navigation control that shows values of several fields from different database tables.

One very important structure in the system is the hierarchy of *records* and *record lists*. A record is one row in a database table and a record list is a list of such records. In S|4|Rights-applications one can navigate through the database with the help of the navigation control that shows the tree hierarchy build from the records and record lists. In S|4|Sales the root of the navigation control can be, for example, a list of contracts, like shown in Figure 17. Every contract in the list contains a list of films and a list of seasons. Every season has a list of episodes, which are the last visible details in the navigation control. In Figure 17 the record lists are marked with bold text.

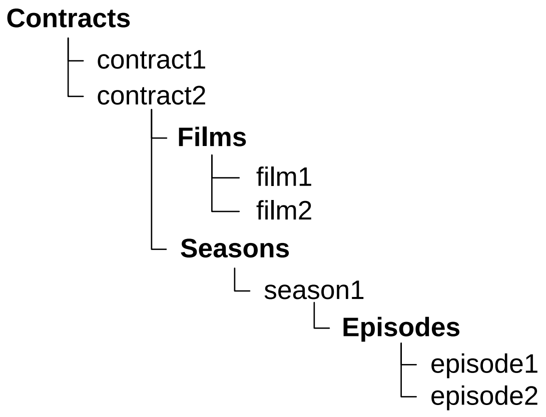
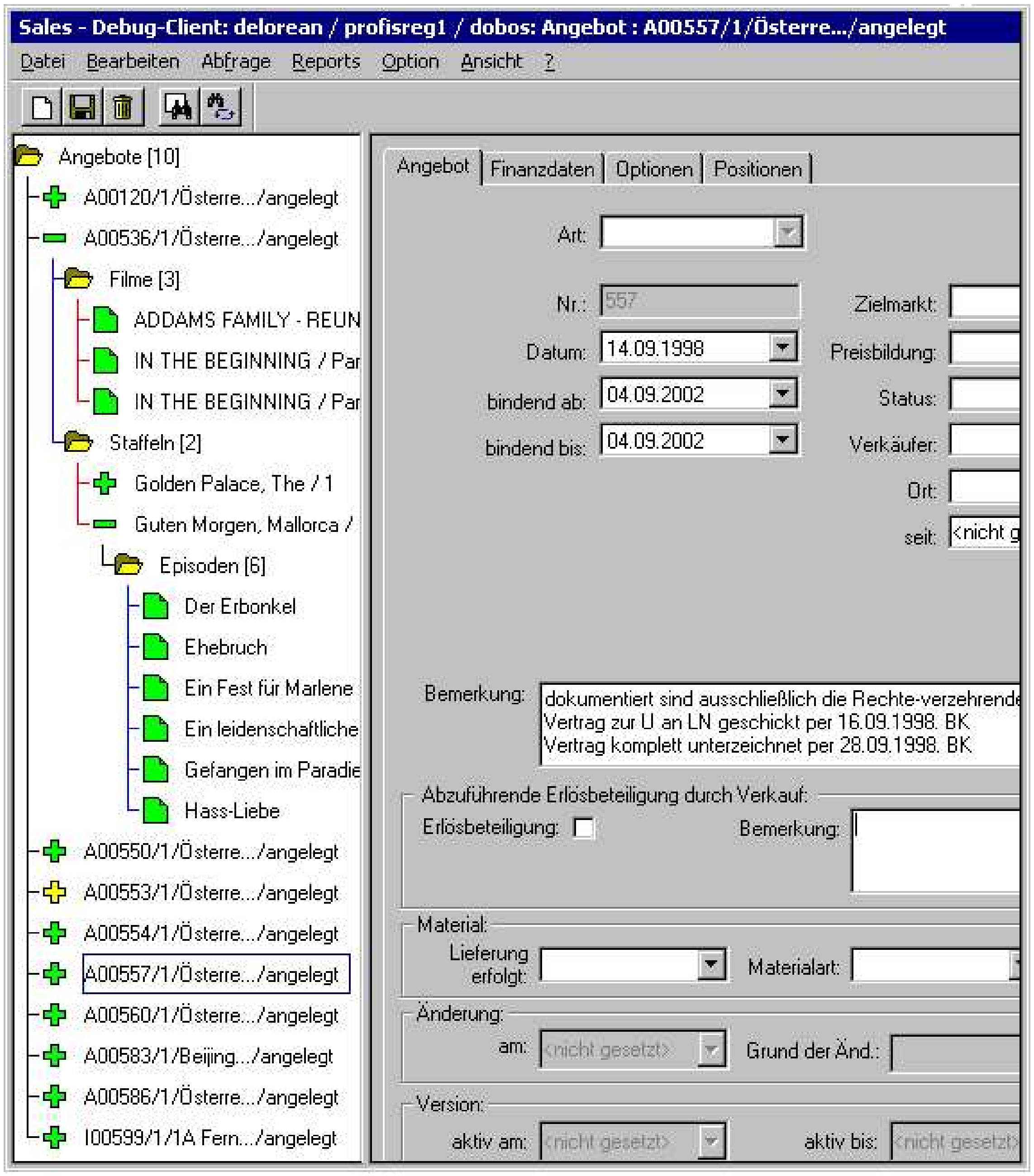


Figure 17: The structure of records and record lists.

Figure 18 shows a typical view of the S|4|Sales-application. The navigation control is in the left part of the window. When the user selects a record or a record list in the navigation control, detailed information about the specific object is shown in the right part of the window.

In subsection 5.5.1 the old navigation tree in S|4|Rights is introduced. This is an example of a solution that bounds the user interface and the application so tightly



Sales - Debug-Client: delorean / profisreg1 / dobos: Angebot : A00557/1/Österre.../angelegt

```
Angebote [10]
├ A00120/1/Österre.../angelegt
├ A00536/1/Österre.../angelegt
│ ├ Filme [3]
│ │ ├ ADDAMS FAMILY - REUN
│ │ ├ IN THE BEGINNING / Par
│ │ └ IN THE BEGINNING / Par
│ └ Staffeln [2]
│   ├ Golden Palace, The / 1
│   └ Guten Morgen, Mallorca /
│     └ Episoden [6]
│       ├ Der Erbonkel
│       ├ Ehebruch
│       ├ Ein Fest für Marlene
│       ├ Ein leidenschaftliche
│       ├ Gefangen im Paradie
│       └ Hass-Liebe
├ A00550/1/Österre.../angelegt
├ A00553/1/Österre.../angelegt
├ A00554/1/Österre.../angelegt
├ A00557/1/Österre.../angelegt
├ A00560/1/Österre.../angelegt
├ A00583/1/Beijing.../angelegt
├ A00586/1/Österre.../angelegt
└ I00599/1/1A Fern.../angelegt
```

Art:
Nr.: 557
Datum: 14.09.1998
bindend ab: 04.09.2002
bindend bis: 04.09.2002
Zielmarkt:
Preisbildung:
Status:
Verkäufer:
Ort:
seit: <nicht g

Bemerkung: dokumentiert sind ausschließlich die Rechte-verzehrend
Vertrag zur U an LN geschickt per 16.09.1998. BK
Vertrag komplett unterzeichnet per 28.09.1998. BK

Abzuführende Erlösbeteiligung durch Verkauf:
Erlösbeteiligung: ☐ Bemerkung:

Material:
Lieferung erfolgt: Materialart:

Änderung:
am: <nicht gesetzt> Grund der Änd.:

Version:
aktiv am: <nicht gesetzt> aktiv bis: <nicht gesetzt>

Figure 18: A typical view on the user interface in S|4|Rights.

together that the separation of the user interface is in practice impossible. Sections 5.5.2, 5.5.3 and 5.5.4 introduce the new navigation control that corresponds to the user interface architecture in S|4|Rights introduced in section 5.1. The new solution can easily be reused and extended and its appearance on the user interface can be changed with only local modifications.

5.5.1 The old navigation tree

The structure of the old version of the navigation tree is illustrated in Figure 19. The base document class has a member object `CResultStructureTbl`, which contains information about the records and record lists that are or have been at some time illustrated in the navigation tree. The `CResultStructureTbl` maintains pointers to the records or record lists it represents and additional information, for example, about the caption of the items in the navigation tree and whether they are selected, visible or expanded.

In S|4|Rights the navigation tree is included in a navigation view. The navigation view class has a member object `CTreeView`, which provides the appearance of the navigation tree.

The functionalities of the navigation tree, like adding or removing a record, or updating the caption of a record, are implemented in the navigation view class. These methods are protected, but the programmer can access them by generating specific event messages. The events are processed by an event handler, which is a procedure that performs a set of actions based on the name of the event it receives.

Some examples of how the old navigation tree is used are presented in Figure 20. The facts that the `CResultStructureTbl` must be maintained ”by hand” and the navigation view must be separately informed about the changes made in the `CResultStructureTbl`, shift a big responsibility onto the programmer and, while being a complex task, provide a source for numerous errors in the system. The lack of proper architecture in the navigation tree of S|4|Rights leads also to repeated code in many places of the application.

To be able to separate the user interface from the application, some fundamental

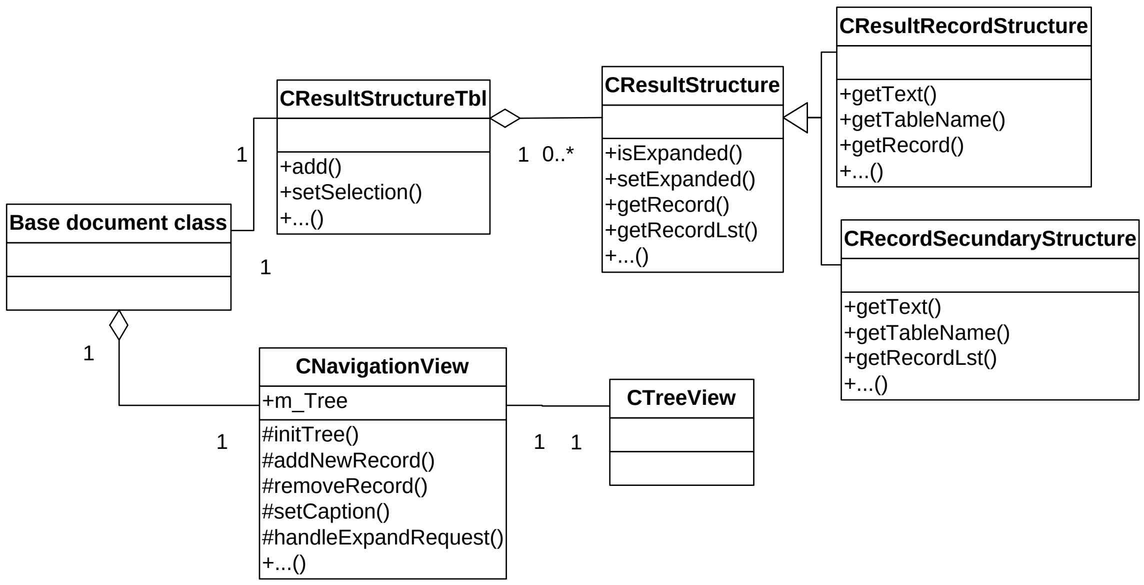


Figure 19: The structure of the old navigation tree.

changes are needed in the navigation tree. In the old navigation tree, for instance, the windows keyboard and mouse notification messages are handled in the navigation view. The class responsible for the appearance on the user interface, `CTreeView`, should be separated from the navigation view. Moreover, the navigation view should have as little responsibility of the functionality of the navigation tree as the other views have responsibility of the interaction controls displayed in them.

In the next subsections the new navigation control is introduced.

5.5.2 Analysis of the navigation control

The goal is to develop a new solution for the navigation control that is encapsulated from the application data and, therefore, reusable. The solution should provide a clear interface between the user interface and the application for supporting separability.

The programmer should be able to use the navigation control over an interface without knowing anything about the implementation of it. The class structure of the control should also enable its easy modification and extension. The layout and the operations of the navigation control for the end user should remain the same as in the

1. Adding a new record:
    (a) find the right index in the `CResultStructureTbl` for adding the new record
    (b) add the new record to the `CResultStructureTbl`
    (c) inform the navigation view about this change by sending the message `ADDNEW`
2. Updating the caption of a record:
    (a) ask the index of the record from the `CResultStructureTbl`
    (b) send message `TEXT_CHANGED` to the navigation view
3. Removing a record:
    (a) ask the index of the record from the `CResultStructureTbl`
    (b) send message `DELETE` to the navigation view
    (c) remove the record from the `CResultStructureTbl`

Figure 20: Typical use of the old navigation tree.

old navigation tree, except for some extensions in the functionality.

In the following, functional requirements from the end user's point of view are listed. Mandatory requirements are that the user should be able to

1. select or unselect one or more nodes or leafs in the navigation control.
2. expand or collapse the descending structure of a node in the navigation control.
3. set or remove a bookmark for one or more nodes or leafs.
4. use both mouse and keyboard to operate with the navigation control.

Optionally the user could be able to

5. change the visibility of some details in the navigation control.
6. change the location of a node or a leaf in the navigation control.

The functional requirements for the navigation control from the programmers' point of view are partly the same, because the programmer must also be able to operate with the navigation control inside the system. The programmer needs, however, some additional functionality that is not accessible for the end user. The additional mandatory requirements are that the programmer should be able to

1. initialise the navigation control.
2. set a node or leaf visible or invisible in the navigation control.
3. inquire whether a node or a leaf is selected, expanded or visible in the navigation control and whether it has a bookmark.
4. get a pointer to the data object behind a node or a leaf selected by the end user.
5. get a pointer to the data object behind a node or a leaf with a bookmark.

A couple of constraints for the first version of the navigation control are:

1. The navigation control can only be used for records and record lists.

2. A record or a record list can be visible in only one navigation control at a time.

For the current use of the navigation control in the modules of S|4|Rights these constraint do not cause any problems but simplify the concept. In the future the navigation control can be extended by removing these constraints.

Key factors related to the navigation control are the end user and the programmer, because they use the control, and the record and the record list, because they are presented in the navigation tree. In Figure 21 below is the analysis class diagram of the navigation control that shows the relations of the key factors.

In the next subsection the chosen solution for the navigation control is presented.

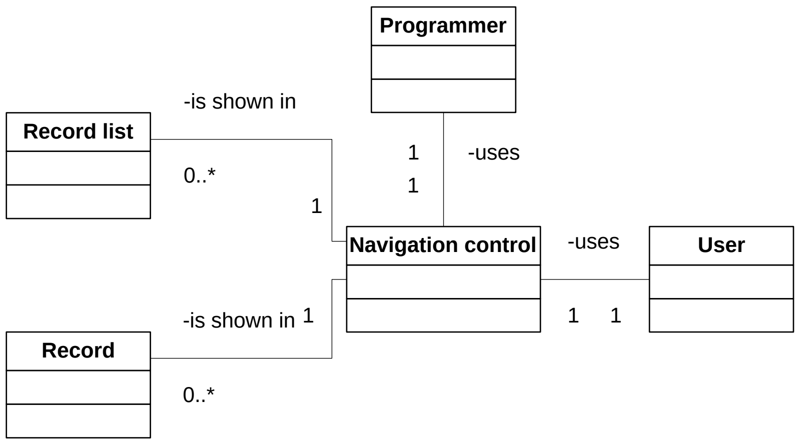


Figure 21: The analysis class diagram for the navigation control.

5.5.3 Design of the navigation control

The structure that is chosen for the navigation control is similar to that of the other interaction controls in S|4|Rights. The navigation control, called `CCtrlNavTree`, has two base classes: `CCtrlNavBase` and `CNavTreeWnd` (see the design class diagram in Figure 22). Like `CCtrlBase` for the other interaction controls, `CCtrlNavBase` is the application interface for the navigation control and `CNavTreeWnd` is responsible for the presentation of the user interface and communication with the user.

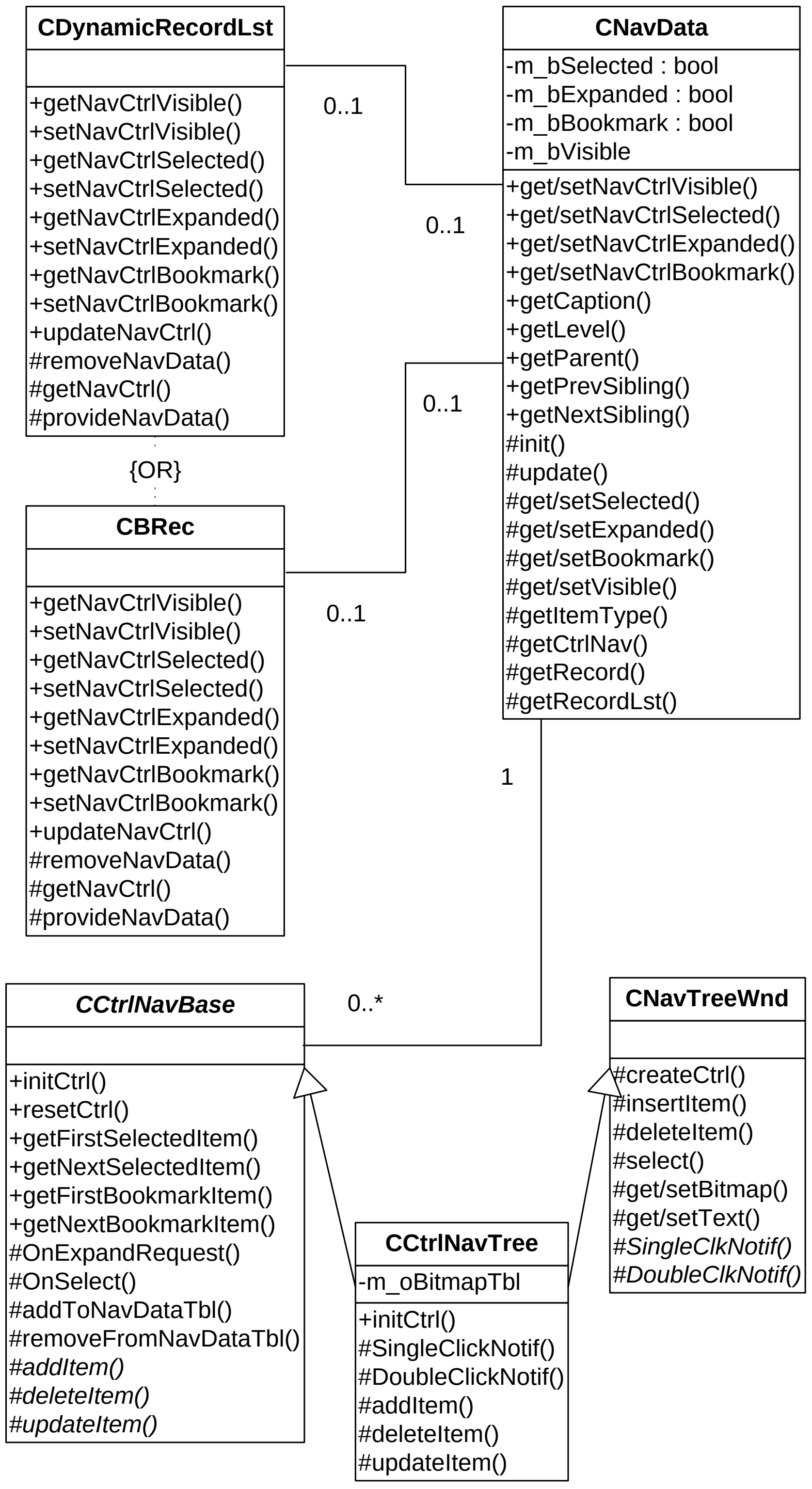


Figure 22: The design class diagram for the navigation control.

The navigation control is an independent interaction control and can be used separately from the control hierarchy introduced in section 5.2. In the future the `CCtrlBase` can also be used as a base class for the `CCtrlNavBase` to obtain a complete hierarchy of the interaction controls.

The existing hierarchy of records and record lists is used to gain the needed information to create and maintain the navigation control. These classes are connected to the navigation control using a class named `CNavData` that is placed between the navigation control and the record and record list classes. Each record and record list has a member pointer object `m_poNavData` that points to the corresponding `CNavData`-object, if that exists. The `CNavData`-object will be created, when the record or record list comes visible for the first time in the navigation control.

`CNavData` contains the information about the record or record list that is essential for the navigation control: the status whether the record or record list is visible, selected or expanded, and whether it has a bookmark in the navigation control or not. `CNavData` also has a pointer to the corresponding record or record list and to the navigation control (`CCtrlNavBase`) it is displayed in.

Using the interface `CCtrlNavBase` is not the only way for the programmer to use the navigation control. The record and record list classes also contain a few methods for handling the navigation control. This way the programmer can, for example, expand a specific record in the navigation control with one function call without having to know any details about the navigation control.

The sequence diagrams in Figures 23 and 24 show how the different objects communicate with each other when the user expands or collapses a record or a record list in the navigation control. The programmer can expand or collapse a record or record list using the function `setNavCtrlExpanded` (see the sequence diagrams).

5.5.4 Validation of the navigation control

The old navigation tree was used in S|4|Rights in the navigation view as mentioned in 5.5.1. A big part of the functionality of the navigation tree was also implemented in the navigation view class. The new navigation control is used in the navigation view

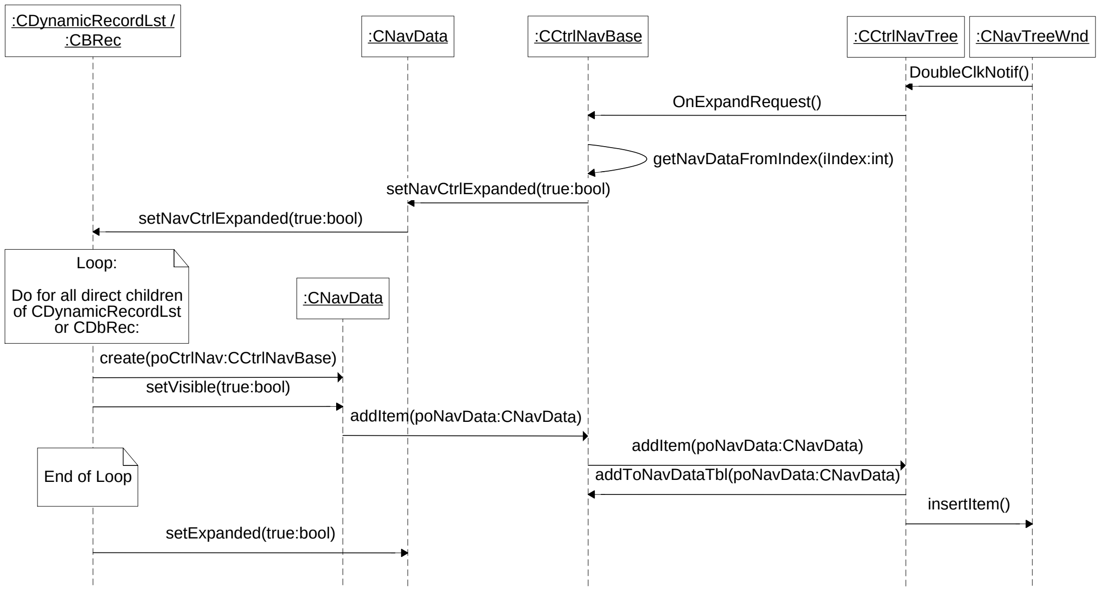

:CDynamicRecordLst / :CBRec
:CNavData
:CCtrlNavBase
:CCtrlNavTree
:CNavTreeWnd
DoubleClkNotif()
OnExpandRequest()
getNavDataFromIndex(iIndex:int)
setNavCtrlExpanded(true:bool)
setNavCtrlExpanded(true:bool)
Loop:
Do for all direct children of CDynamicRecordLst or CDbRec:
:CNavData
create(poCtrlNav:CCtrlNavBase)
setVisible(true:bool)
addItem(poNavData:CNavData)
addItem(poNavData:CNavData)
addToNavDataTbl(poNavData:CNavData)
End of Loop
insertItem()
setExpanded(true:bool)

Figure 23: A sequence diagram illustrating expanding a record or record list in the navigation control.

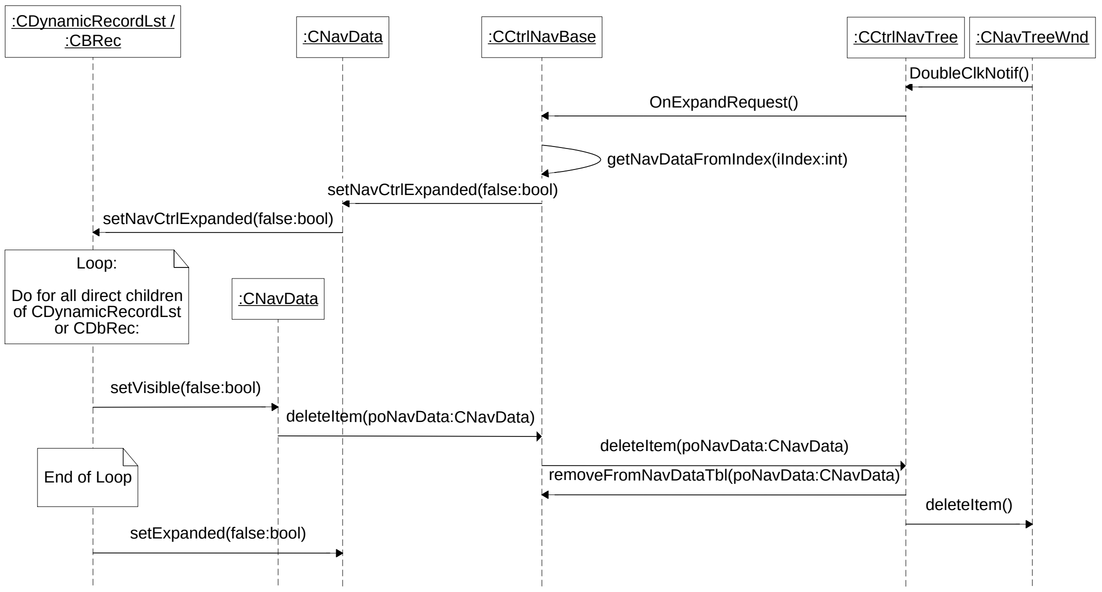

:CDynamicRecordLst / :CBRec
:CNavData
:CCtrlNavBase
:CCtrlNavTree
:CNavTreeWnd
DoubleClkNotif()
OnExpandRequest()
getNavDataFromIndex(iIndex:int)
setNavCtrlExpanded(false:bool)
setNavCtrlExpanded(false:bool)
Loop:
Do for all direct children of CDynamicRecordLst or CDbRec:
:CNavData
setVisible(false:bool)
deleteItem(poNavData:CNavData)
deleteItem(poNavData:CNavData)
removeFromNavDataTbl(poNavData:CNavData)
End of Loop
deleteItem()
setExpanded(false:bool)

Figure 24: A sequence diagram illustrating collapsing a record or record list in the navigation control.

too, but its functionality is encapsulated inside the navigation control classes and it can, therefore, be used as an independent interaction control in the system.

It is much easier for the programmer to use the new navigation control than the old one. For a comparison, in Figure 25 some examples of how to use the navigation control are shown. These are the same operations that were introduced for the old navigation view in Figure 20.

---

1. Adding a new record:
   - Make a function call

     `poRecord->setNavCtrlVisible(true)`
2. Updating the caption of a record:
   - Make a function call

     `poRecord->updateNavCtrl()`
3. Removing a record:
   - Make a function call

     `poRecord->setNavCtrlVisible(false)`

Figure 25: Typical use of the new navigation control.

While adding a new record, the programmer does not have to worry about the position of the record in the navigation control; it will be placed automatically. For this purpose, the navigation control uses the record and record list structure in S|4|Rights. The appearance of the navigation control also changes automatically and, therefore, the programmer does not have to inform the navigation view about the changes, like with the old navigation tree.

Adding and removing a record from the navigation control can be done by calling the function `setNavCtrlVisible`. The parameter determines, whether the record (or record list) will be added or removed in the navigation control. When a record is removed, the children of the record are also removed. Similarly, when a record list is added, its parent record list is also added automatically if needed. This kind of automatic behaviour of the navigation control reduces the work needed by the programmer and helps avoiding errors.

These are only a couple of examples of the methods available for the programmer in the new navigation control. They are introduced to show how easily the programmer can now communicate with the navigation control.

All mandatory requirements mentioned in 5.5.2 are fulfilled in the new navigation control. As the development continues, the optional requirements will also be realized. Non-functional requirements for the navigation control mentioned in subsection 5.5.2 were that the control should be encapsulated from the application data, reusable, easy to modify and extend, and it should provide a clear interface between the user interface and the application.

The *encapsulation from the application data* is achieved through the concentration of the functionality into the navigation control classes `CCtrlNavBase`, `CCtrlNavTree` and `CNavTreeWnd`. This and the fact that the application data structure presented in the navigation control is separated from the control with help of the `CNavData`-class, make the navigation control *reusable*.

How the architecture of the navigation control supports the *easy modification and extension* of the control is already shown for the other interaction controls in section 5.4. In the same way, new navigation controls with a different appearance can be developed and used without changing the application, as long as the new navigation controls are inherited from the `CCtrlNavBase`, the base class for the navigation controls.

`CCtrlNavBase`, the application interface of the navigation control, provides the *clear interface between the user interface and the application.* `CCtrlNavBase` has no information about the application data; it only has a vector of `CNavData`-pointers. The `CNavData`-class hides the application data from the navigation control. This, in addition to separating the user interface from the application, also promotes reuse of the navigation control. In case that in the future one wants to use the navigation control for some other objects than records and record lists, changes are needed only in the `CNavData`-class and in the classes that are to be presented in the navigation control.

One constraint for the navigation control is that the control can only be used for records and record lists. Like mentioned above, to remove this constraint, changes are needed in the `CNavData`-class. The first step in this direction could be creating a structure of navigation data classes. `CNavData` would be the base class declaring the necessary methods and implementing the common ones. For the objects presented in the navigation control the programmer would then create an own navigation data class, inherited from the `CNavData`.

For the current use two classes would be inherited from the `CNavData`-class: `CNavDataRecord` and `CNavDataRecordList` that implement the needed methods for records and record lists, like `getParent` or `getLevel`. In addition, those methods in record and record list classes that are used for communicating with the navigation control (see Figure 22) have to be implemented in the object class presented in the navigation control.

The record and record list classes presented in the navigation control belong to a hierarchy of different kind of record classes and other database related classes. On an upper level in the hierarchy these two classes have a common base class. At first sight it would make sense to declare the needed methods in the base class and override them in the inherited classes `CBRec` and `CDynamicRecordList`. However, this solution was not chosen, because the hierarchy includes many classes for which these methods are irrelevant. Adding the methods in the base class would also make it difficult to keep the solution of the navigation control compact.

One possibility would be to add a new virtual base class only for the two classes `CBRec` and `CDynamicRecordList`. This virtual class would declare the mandatory methods as pure virtual methods, so that the programmer would be advised to implement them in the record and record list classes. The same virtual base class could be used for other objects presented in the navigation control.

To be able to remove the second constraint, so that a record or record list could be shown in many navigation controls at the same time, small changes have to be done in the record and record list classes. Instead of one pointer to a `CNavData`-object, the record and record list classes should have a vector of navigation data objects, one for each navigation control where they are being presented. The methods for the communication with the navigation control all need an additional parameter that determines the navigation control where the record should be set visible, expanded etc.

5.6 Validation of the user interface architecture of S|4|Rights

At the beginning of chapter 3 arguments for the separation of the user interface and the application were listed. In this section these arguments are used to verify whether the user interface architecture of S|4|Rights supports the separability or not.

Changes to the user interface do not cause changes to the application, and vice versa: The sufficient functionality in the application interface of the user interface architecture in S|4|Rights makes this possible: in the application part only functions of the application interface are used without calling functions in the user interface classes directly, and vice versa. The encapsulation and the division of the user interface into layers make the changes local, and reduces, therefore, *the impact of change* and *the cost of maintenance.*

The application becomes portable: Using the new user interface architecture in S|4|Rights, the changes needed for being able to execute the application on another platform are local. The presentation of the user interface is encapsulated in the presentation part from the rest of the user interface and can be replaced by making changes only in the dialogue control part, which uses the functions of the presentation part.

The components of the software system become reusable: This advantage is clearly achieved in the user interface architecture of S|4|Rights. Using the abstract base classes in the application interface of the user interface architecture, like `CCtrlBase` for the interaction controls, makes the code reusable, because it will not be bound to specific user interface classes. One achievement from this point of view was making the navigation control reusable.

It is possible to implement multiple user interfaces for one application: Changing the appearance of the interaction controls is easy in the new user interface architecture. Using interaction control families (see section 5.4.3), different appearances for the interaction controls can be implemented. After that changes are only needed in the resource files that define the appearance of the user interface.

Customisation of the application becomes easier: The easy modification of the user interface is the key to the easy customisation (see the previous paragraph).

The cooperation of user interface designers and application developers is easier: Because of the separation of the presentation and functionality of the user interface, changes to these parts can be made separately. This way the user interface designers and application developers can easier share their work.

Fast and easy modification of the user interface code and layout is possible: The examples introduced in sections 5.4 and 5.5 show that modifying and extending the user interface architecture in S|4|Rights is easy and leads to logical and well-structured solutions.

The program code becomes more readable: The logical structure and clear separation of responsibilities between the different parts of the user interface architecture help developers to understand the code in S|4|Rights and easily find the right way to use the user interface functionality. The encapsulation plays a very important role in making the code readable. Because of the common interface for the interaction controls, the developers only need to know how to use one interaction control and then, using the same methods, use all of them.

Different programming languages can be used for the implementations of the application and the user interface: The use of different programming languages for the implementation was yet not needed in S|4|Rights and, therefore, has not been tested.

The lifetime of the application gets longer because of the increase in flexibility: This is, of course, an important goal in software development. The new user interface architecture in S|4|Rights make also the future changes easier to make and, therefore, extends the lifetime of the application.

The *decrease of performance* is the usually argumented disadvantage of the separation of the user interface and application. Because the modules in S|4|Rights use databases with huge amounts of information, the database searches are the crucial factor in the performance. The possible little change in the performance because of the modularisation does not make the difference. In some cases it even increased the performance, because restructuring the user interface revealed some parts of the user interface code that made unnecessary database searches.

Another common problem that comes with the separation is the difficulty of *connecting the application data to the user interface* without disturbing the separation. This is solved in S|4|Rights using class interfaces that make the connection between the two parts well-defined and local.

6 Conclusions

The main emphasis of this work was to find out, what kind of an architecture should an interactive software system have so that the user interface and the application can be separated from each other. I suppose there is no single answer to this question. There are different kind of software architectures that suite for different kind of interactive systems. All of these architectures, anyhow, have some common properties like the division into layers or the important meaning of the application interface.

This thesis introduces some workable solutions that as part of a suitable user interface architecture support the separation of the user interface and the application. These suggestions include ensuring sufficient functionality in the application interface, using abstract base classes in the application interface and using multiple inheritance in separating the presentation, dialogue control and application interface from each other.

An additional question to be answered in this work was to find a structure for the interaction controls that enables changes on the user interface without affecting the business logic of the application. This was achieved in S|4|Rights by using the structure in the user interface architecture also for the interaction controls. An additional question was to find a way to connect the application data to the user interface without disturbing the separability. The application data in S|4|Rights is made available for the user interface using semantic objects as values in the interaction controls. The abstract base classes of these objects are one part of the application interface. This way the application data is connected to the use interface in a well encapsulated way and do not disturb the separability.

One practical goal of this work was to develop a new navigation control for the S|4|Rights modules that utilizes the chosen structure for the interaction controls. The navigation control is already actively used in eight different modules of the software package S|4|Rights. While adding the navigation control in the single modules, its implementation was every time slightly improved. The experience gathered using the new navigation control showed that the structure is workable and can be used in different situations as well.

The application interface of the interaction controls is only one part of the application interface in the user interface architecture of S|4|Rights (see section 5.1). The abstract base classes for frame windows, views, pages, dialogs etc. also belong to it. One of the next tasks in developing the user interface architecture in S|4|Rights is to create a structure for the dialogs, much like the structure of the interaction controls. Work must still be done before the user interfaces of the modules in S|4|Rights are completely separable, but this thesis is a significant step forward into the right direction.

References

[1] *A Metamodel for the Runtime Architecture of an Interactive System,* The UIMS Tool Developers Workshop, SIGCHI Bulletin, Vol. 24, Issue 1, pp. 32-37, (1992).

[2] P. S. C. Alencar, D. D. Cowan, C. J. P. Lucena, L. C. M. Nova: *Formal Specification of Reusable Interface Objects,* ACM SIGSOFT Software Engineering Notes, Proceedings of the 17th international conference on software engineering on Symposium on software reusability, Vol. 20, Issue SI, (1995).

[3] C. Alexander: *A Pattern Language,* Oxford University Press, (1977).

[4] J. L.Alty, P. MnKell: *Application Modelling in a user interface Management System,* in: E. Edmonds (ed.): The Separable User Interface, Academic Press, pp. 297-311, (1992).

[5] S. Burbeck: *Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC),* http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, (1992). Referred 19.11.2002.

[6] K. A. Butler, R. J. K. Jacob, B. E. John: *Human-Computer Interaction: Introduction and Overview,* Proceedings of the conference on CHI 98 summary : human factors in computing systems, ACM Press, (1998).

[7] K. Coutaz: *PAC, an Object Oriented Model for Dialog Design,* Human-Computer Interaction - INTERACT'87 proceedings, H. J. Bullinger and B. Shackel (eds),Elsevier Science Publishers B. V., pp. 431-436, (1987).

[8] C. Calvary, J. Coutaz, L. Nigay: *From Single-User Architectural Design to PAC*: a Generic Software Architecture Model for CSCW,* conference proceedings on Human factors in computing systems, ACM Press, (1997).

[9] C. A. Constantinides, A. Bader, T. H. Elrad, M. E. Fayad, P. Netinant: *Designing an Aspect-Oriented Framework in an Object-Oriented Environment,* ACM Computing Surveys (CSUR), Vol. 32, Issue 1es, (2000).

[10] 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, Academic Press, pp. 213-225, (1992).

[11] A. Dix, J. Finlay: *Readings in Human-Computer Interaction: Toward the Year 2000,* Morgan-Kaufman, (1995).

[12] E. Edmonds: *The Emergence of the Separable User Interface,* in: E. Edmonds (ed.): The Separable User Interface, Academic Press, pp. 5-18, (1992).

[13] E. Edmonds, E. McDaid: *An Architecture for Knowledge-based Front Ends,* in: E. Edmonds (ed.): The Separable User Interface, Academic Press, pp. 263-269, (1992).

[14] M. Evers: *A Case Study on Adaptability Problems of the Separation of User Interface and Application Semantics,* CTIT Technical Report TR99-14, Centre for Telematics and Information Technology, (1999).

[15] G. Enderle: *Report on the Interface of the UIMS to the application,* in: E. Edmonds (ed.): The Separable User Interface, Academic Press, pp. 285-293, (1992).

[16] J. Foley, A. van Dam: *Fundamentals of interactive computer graphics,* Addison-Wesley, Systems Programming Series, (1984).

[17] J. D. Foley: *Future Directions in User-Computer Interface Software,* ACM SIGOIS Bulletin, Conference proceedings on Organizational computing systems, Vol. 12, No. 2-3, (1991).

[18] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software,* Addison-Wesley, Professional Computing Series, (1995).

[19] M. Green: *Report on Dialogue Specification Tools,* in: E. Edmonds (ed.), The Separable User Interface, Academic Press, pp. 211-226, (1992).

[20] H. R. Hartson, D. Hix: *Human-Computer Interface Development: Concepts and Systems for Its Management,* ACM Computing Surveys, Vol. 21, No. 1, (1989).

[21] T. Hewett: *Curricula for Human-Computer Interaction,* ACM Special Interest Group on Human-Computer Interaction Curriculum Development Group, (1992).

[22] W. D. Hurley, J. L. Sibert: *Modelling User Interface-Application Interactions,* in: E. Edmonds (ed.), The Separable User Interface, Academic Press, pp. 151-165, (1992).

[23] J. Johnson: *Selectors: Going Beyond User-Interface Widgets,* Conference proceedings on Human factors in computing systems, ACM Press, (1992).

[24] J. Kuusela: *Architectural Patterns,* Nokia Research Center, http://www.cs.hut.fi/˜jku/slides/lect2/index.html, (1997). Referred 19.11.2002.

[25] J. M. Manheimer, R. C. Burnett, J.A.Wallers: *A Case Study of User Interface Management System Development And Application,* ACM SIGCHI Bulletin, Conference proceedings on Human factors in computing systems: Wings for the mind, Vol. 20, Issue SI, (1989).

[26] T. Moran: *A framework for studying human-computer interaction,* in: R. A. Guedij et al (ed.): Methodology of Interaction, pp. 293-302, (1980).

[27] Microsoft Corporation: *Microsoft Developer Network Library,* (2001).

[28] B. Myers, S. E. Hudson, R. Pausch: *Past, Present, and Future of User Interface Software Tools,* ACM Transactions on Computer-Human Interaction, Vol. 7, No. 1, 3-28, (2000).

[29] H. Ossher, P. Tarr: *Using Multidimensional Separation of Concerns to (Re)shape evolving Software,* Communications of the ACM, Vol. 44, Issue 10, (2001).

[30] A. Prat, J. Lores, P. Fletcher, J. M. Catot: *Back-End Manager: An Interface between a Knowledge-based Front End and its Application Subsystems,* in: E. Edmonds (ed.): The Separable User Interface, Academic Press, pp. 273-280, (1992).

[31] D. Riehle: *How and Why to Encapsulate Class Trees,* ACM SIGPLAN Notices, Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications, Vol. 30, Issue 10, pp. 251-264, (1995).

[32] B. Stroustrup: *Komponenten,* in: B. Stroustrup: Die C++ Programmiersprache, Addison-Wesley, pp. 812-821, (1997).

[33] P. Szekely: *Implementation of a Program's Designer Model,* in: E. Edmonds (ed.): The Separable User Interface, Academic Press, pp. 315-330, (1992).

[34] M. Zhou, K. Houck: *A Semantic Approach to the Dynamic Design of Interaction Controls in Conversation Systems,* Proceedings of the 7th international conference on Intelligent user interfaces, ACM Press, pp. 167-174, (2002).