T U M

I N S T I T U T F Ü R I N F O R M A T I K

Component-Oriented Redesign of the CASE-Tool AutoFocus

Klaus Bergner, Franz Huber,
Andreas Rausch, Marc Sihling

TUM-I9752
Dezember 1997

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-12-1997-I9752-100/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Component-Oriented Redesign of the CASE-Tool AutoFocus*

Klaus Bergner, Franz Huber, Andreas Rausch, Marc Sihling

Institut für Informatik
Technische Universität München
D-80290 München
http://www4.informatik.tu-muenchen.de

15th December 1997

Abstract

In this paper, we explore how the componentware paradigm can be used to re-engineer existing software systems. As a case study, we use the prototype implementation of the CASE tool AutoFocus which has been developed using standard object-oriented design techniques. Although designed for re-usability with respect to certain criteria, AutoFocus did not use a component-based approach yet. The case study concentrates on the repository subsystem, as this part is currently the least "modular" part of AutoFocus.

We outline essential concepts of the componentware paradigm, including a methodology how to carry out a component-based design process, introduce the current state of AutoFocus, and sketch how we derive a component-based redesign, including a migration strategy from purely object-oriented legacy systems to component-based systems.

Keywords: Componentware, Object-Oriented Software Engineering, Design, Redesign, Java, CASE, Repository

*This paper originated in the ForSoft project A1 on "Component-Based Software Engineering", supported by Siemens ZT, and in the sub-project A6 of the "Sonderforschungsbereich (SFB) 342" supported by the DFG.

Contents

1 Introduction

Componentware carries the old dream of assembling software systems like buildings from a set of given components to undreamt brilliancy.

In architecture, the customer first tells the architect his functional and non-functional requirements in an informal way, for example, the number and function of rooms and the money he or she wants to spend. Then the architect constructs a first, rough ground-plan and several side-views. If this "prototype" fits with the customer's visions, the architect specifies a more detailed and technical construction plan. This plan describes the different components of the building and their relationships, like walls and windows, and how they fit together. Now the architect puts these components up for tender. At last, the "best" component producer will get the job, place the components to the architect's disposal and integrate them into the building. Over the whole process the architect's construction plan is the communication plattform for all parties working on the building.

The componentware software-engineering process should be similar: During analysis, the software architect constructs a rough analysis model and component-based prototyps to fathom and finally fulfil the customers visions. Then, in an iterative design process he will refine the analysis model until the essential components are found. Keep in mind that this is not a pure top-down process but rather a constant switch between top-down and bottom-up because the architect has existing and given target components in his mind during refinement. The result of the design process is a mostly complete construction plan for the software system. Now, eventually different component vendors can ship the components. The software architect has to take care of the components quality, check whether they fulfil the requirements, and finally assemble them to build the complete software system.

The most important points during this process are the separation of the two roles *software architect* and *component developer* and an excellent construction plan.

However, today developing a complete new software system is not the normal case. In most projects one has more or less to reengineer, integrate and reuse legacy systems or at least special parts of such systems. This necessitates adapting the componentware software-engineering process described above. There are several kinds of processes and techniques for component-oriented redesign of legacy systems [DPB96, MLB95].

Despite this fact, the redesign process we introduce in this paper is close to the vision of componentware software-engineering we described above. First, we present a closer look at AutoFocus from the user's view in Section 2. In Section 3 we discuss the basic concepts and ideas of componentware. The old legacy architecture of AutoFocus is then presented in Section 4. The next section contains our migration strategy; the architecture of the old AutoFocus is analyzed and useful, reusable components are worked out. Finally, the new, component-oriented architecture of AutoFocus is developed in Section 6 according to our vision of the componentware software-engineering process. A short conclusion ends the paper.

2 The Project: Redesigning AutoFocus

AutoFocus is a prototype implementation of a multi-user software engineering tool for the specification and simulation of distributed systems based on the formal development method FOCUS [BDD+92]. Within this framework, AUTOFOCUS provides graphical description formalisms covering different aspects of distributed systems:

- System structure diagrams (SSDs) describe the static structure of a system as a network of interconnected components exchanging messages over channels,
- State transition diagrams (STDs, or automata) describe the behaviour of system components, and
- Extended Event Traces (EETs, basically a subset of Message Sequence Charts as standardized in ITU Z.120 [IT93]) specify the dynamic interactions between components during system runs.

For each of these formalisms, AUTOFOCUS provides its own graphical editor (see Figure 1). The conceptual basics of AUTOFOCUS and its description techniques are described in detail in [HSS96] and [HSSS96].

AUTOFOCUS is implemented as a client/server system: It consists of a central server that manages the repository with the development documents and provides mechanisms for client access control and version management. The development documents in the repository are organized according to projects.

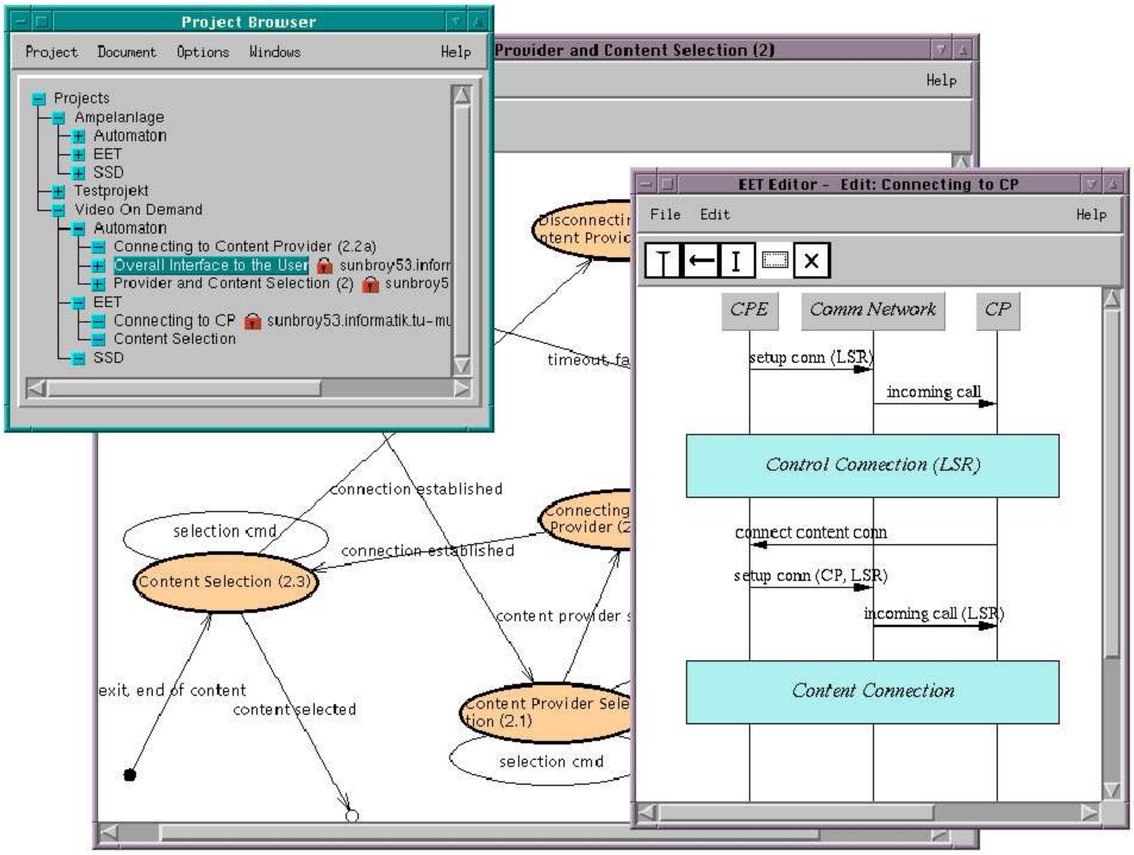

Figure 1: Project Browser, STD Editor, and EET Editor of the AUTOFOCUS Client

The AUTOFOCUS client (see Figure 1) contains a project browser used to navigate through the development projects and documents in the repository, and the graphical editors necessary to edit the different kinds of diagrams.

Our further plans with AutoFocus are

- to remove its current deficiencies (see Section 4.2),
- to provide support for easy extendability with arbitrary description techniques,
- to allow different working methods for users, and
- to make it a flexible and powerful toolkit for experimentation with new CASE tool concepts.

The design of a generic distributed multi-user CASE toolkit is a very difficult task. Especially the repository mechanisms—among them support for persistence, versioning, locking, and transactions—must be carefully coordinated and harmonized. The requirement of adaptability to various description techniques and working methods suggests the use of a flexible architecture and requires a well-structured software development approach. Component-oriented techniques, as presented in the next section, claim to fulfill such high demands.

3 Componentware: Concepts and Methodology

3.1 Essential Characteristics

Componentware can be defined as software development with the help of components. Its goals are very similar to the goals of object-orientation: Information hiding and decoupling shall lead to well-structured systems consisting of understandable and reusable parts. In contrast to object-orientation, where the basic concepts have settled to some extent during the last years, it tends to be difficult to find a definition for component concepts everyone agrees on—some people consider components to be objects, some other see also monolithic legacy systems or even design patterns as components. Therefore, we will not provide a sharp definition in the following. Instead, we give some characteristics that, in our opinion, hold for the essential approaches.

Analogously to object-orientation, we distinguish the concept of an instance from the concept of a type describing the features common to a set of instances. We deliberately avoid the object-oriented terms "class" and "object" here in order not to mix up the concepts, and usually speak of *component instances* and *component types*. Note that the distinction between component instances and types is unnecessary in some approaches, as, for example, with large legacy components, where only one component instance of a kind exists. However, we introduced the type concept to get a clear and uniform framework of notions suitable also for approaches with explicit types like Java Beans [Javb].

3.1.1 Component Type Characteristics

Self-Describing Export and Import Interfaces The concept of an interface is central to component-orientation. Basically, an interface consists of

- a *signature* part, describing the syntax of a specific functionality and, based on that,
- a *specification* part, describing the component's behaviour in a formal or non-formal manner.

An interface thus comprises the syntax and the semantics of a component type. Although each interface must have a signature, we do not require it to specify the semantical interactions completely by allowing underspecification.

Component types can *implement* resp. *export* one or more interfaces, and they can also *import* interfaces from other component types. Import interfaces specify the functionality needed to implement the exported services of a component type. Together, import and export interfaces specify the interactions between connected component instances of a system (cf. Section 3.2.1). The information provided by component type interfaces can be used by tools for visualization, allowing the composition of component instances in a very comfortable way (cf. Section 3.2).

Language-Independent Access An important step towards better reuse of existing program code (and a major enhancement compared to object-orientation) is the ability of components to communicate with other components programmed in a different language.

Standard Interfaces Most components export a couple of standard interfaces for basic functionality like persistence or configuration management. By implementing standard interfaces, components can easily participate in very powerful mechanisms. In the context of some componentware approaches or of certain component-oriented systems, all components must implement a set of common standard interfaces. Normally, this includes at least a standard interface for querying the signature part of the component's interfaces, e.g. for visualizing a components interface in a tool.

Adaptability Reuse of existing component types requires development techniques for adapting and fitting them. We describe some of the possible adaption techniques in Section 3.2.3.

3.1.2 Component Instance Characteristics

Identity Each component instance can be adressed via a single identifier, which is unique in the considered context.

Data State A component hides its contained data from other components: Communication happens only via its interfaces. The data state is usually persistent, but non-persistent components are also possible.

Customizability During the process of system composition, component instances can be customized for their intended purpose. Although the customization information is part of the data state of a component, it can usually not be changed by users and is, therefore, persistent over all runs of a system. Note that we distinguish between customization and *configuration*: Configuration is done by end users and can be seen as part of the normal functionality of a component or the entire system.

3.2 Development with Components

A very important aspect of componentware is the vision of a new way of programming: Development of a system happens mainly via composition of existing, reusable components.

3.2.1 Connecting Components

Two components can be *connected* if one of them exports an interface that is compatible with an interface imported by the other one. An exact definition of interface compatibility depends on the special componentware approach and the used specification techniques and is therefore omitted here. Connected component instances are visualized as a directed graph where the edges point from the importing to the exporting component, as shown in Figure 2. Following UML [BRJ97], export interfaces are shown as bubbles.

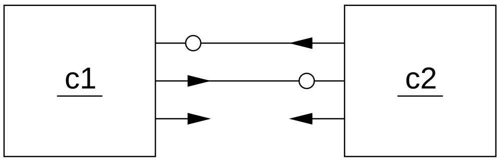

Figure 2: Visualization of Connected Component Instances

There are two possibilities with respect to changes of the component instance connection graph:

- The graph structure may be *static*, meaning that it can only be changed by a system developer.
- The graph structure may be *dynamic*. In this case, the system itself controls the creation and deletion of components and the connections between them.

Static connection structures are a lot easier to comprehend, mostly because they can be easily visualized with graph-like diagrams like the one of Figure 2.

3.2.2 Hierarchical Components

Components can back up on other components to realize their functionality, thus leading to the notion of *hierarchical components*. For a hierarchical component type must be specified

1. on which component instance(s) it relies on,
2. how the contained component instances are connected to each other, and
3. in which way the resulting functionality is achieved.

For example, on the left side of Figure 3 a component instance is refined by three other component instances that are connected to each other. In this view it makes sense to distinguish between *internal* interfaces for communication with the enclosed components and *external* interfaces leading to the outside. Everything between is called the *glue code* (shown in grey in the figure): It connects internal and external interfaces and adds functionality and data specific to the component.

A slightly simpler variation of this schema is shown on the right side of Figure 3: Here, the glue code is concentrated in an own *glue component* which is then connected to the internal and external interfaces of the redefined component. Thus, the glue code has its own container and creating a new component can be done only by connecting lower-level

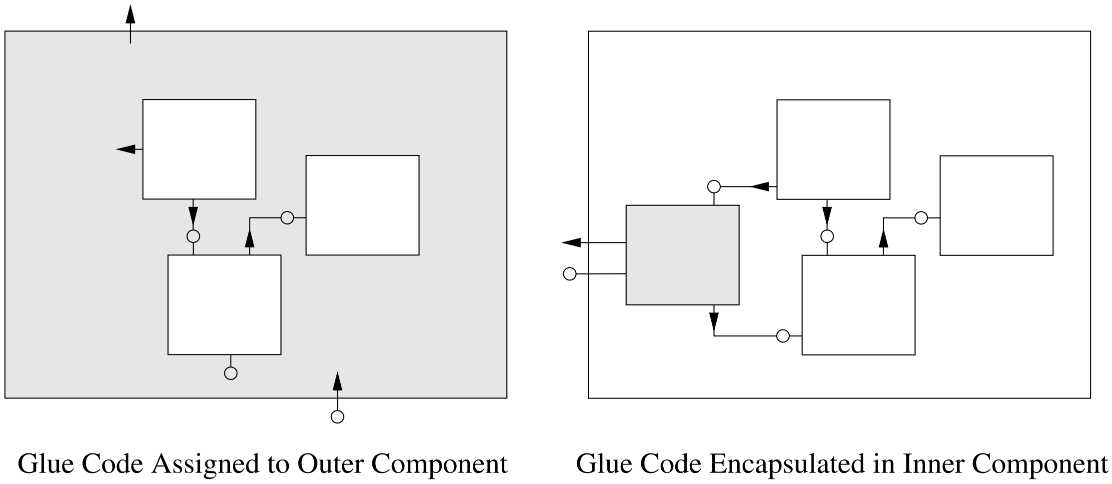

Figure 3: Different Variants of Hierarchical Components

components. This variation basically results in a conceptually flat component structure in which outer components are only design constructs without own functionality.

Most componentware approaches assume that contained components are not visible in the interface of their enclosing component. However, the glue component variant above suggests that this is not strictly necessary: One can also imagine a concept where a component's interface contains some subcomponents or even dynamically changing sets of subcomponents [Ber97].

3.2.3 Adaption of Components

The success of a component can be defined as how often it is reused in various software systems. However, the most difficult part in the construction of a component is to abstract from the requirements of the actual situation and to anticipate its use in other systems. If this is not done carefully, unwanted consequences may arise:

- The component may not be reusable because it was tailor-made for a special application and is, therefore, too specific. Usually, developers have to write a couple of such specific, yet similar components until they realize that a more general, reusable component can be built or used.
- The component is too weak and does not offer sufficient application or customization functionality.
- The component was over-engineered and is too general which means that customization to actual needs is difficult and requires a lot of work.

The art of component development lies between those extremes. Sadly, developers will often be confronted with components that can not be customized for their requirements, necessitating the adaption of the component:

- *Inheritance* has proven worthy in object-orientation to enhance the functionality of a class, and it can in principle be useful also with componentware. However, inheritance leads to a strong coupling between the participating classes and should be used with care. If possible, other, simpler ways of adaption should be chosen.

- A *wrapper* component can be used to shield a reused component from the outside. The wrapper component makes use of the functionality of the wrapped component and offers an adapted interface to other components. Wrappers are a special form of hierarchical components (cf. Section 3.2.2).
- A component may offer a special *adaption interface* that allows other components to engage in certain situations. An example for this technique is an interface that allows another component to register one of its methods as a callback in response to a trigger.
- The most general adaption technique is of course to *adapt the component's code*. However, this requires a thorough understanding of the component's implementation and access to its source code.

It is to hope that the presence of well-designed, standardized components for a certain application domain will make it possible to avoid adaption and to resort only to customization.

3.3 Views and Description Techniques

The main problem in large scale systems lies in reducing their inherent complexity to a manageable and understandable size. Abstraction is a proven concept to achieve this; it allows to build different views onto a system each focusing on special characteristics or structures. We can mainly distinguish two different views on a component, depending whether we look at it as a component user or as a component developer (cf. Section 3.5.1):

- Each component offers at least a *blackbox view*. It contains export and import interfaces, but says nothing about a component's concrete implementation. The blackbox view is, therefore, sufficient for the composition and customization by users of the component: It allows them to find suitable components and to understand their purpose, functionality, usage, and restrictions.
- The *glassbox view* reveals all information about the internal structure of the component. It is necessary for further development and adaption of a component by a component developer.

Each of these basic views may consist of various documents describing different aspects of the properties and the behaviour of a certain component type. Possible are, for example, documents consisting of informal text, graphical techniques like those provided by UML [BRJ97], interface description languages like IDL [COR], or formal specification techniques.

3.4 Classification of Component Types

Classification is helpful for all people involved in the development process to share the same understanding of components. Imaginable are various kinds of classifications which are not necessarily orthogonal to each other. First, we will present a very common one:

Business-Oriented Components model the business-related concepts that are visible and understandable for the system's users. In the context of a CASE tool like

AutoFocus, this amounts to the concepts familiar to distributed systems engineers: Examples are specification documents, document repositories, and single elements of modeling techniques like, for example, the states and transitions of a state transition diagram.

Technical Components are hidden from the user and part of the implementation. An example, again taken from the context of a CASE tool, would be the database component underlying the repository or a component that implements the transaction management.

Another classification focuses on the dynamic instantiation and connection structure of component instances:

Herd Components can be created, deleted, and connected under the control of the system. Herd components model the application's changing data and represent technical concepts like the dialogs of a system's GUI.

Manager Components form the static architecture of a system and fulfill the task of organizing the herd components. An example is a system's dialog manager which keeps track of all actual dialogs. Manager components are often singletons.

3.5 Component-Oriented Methodology

3.5.1 Roles in the Development Process

One of the central issues with componentware is the separation of the roles of component developers and component users. It is a necessary prerequisite for the rise of a market of specialized, reusable high-quality components needed to build the large and highly complex systems of the future. Other, more mature industry branches know this separation for a long time, as can, for example, be seen with the building industry [Hin97]. We expect that the following, specialized roles will evolve in the context of component-oriented software development:

Component Developer: Components are developed by specialized component vendors or by in-house reuse centers in big enterprises. The tasks of a component developer are to recognize the common requirements of many customers or users, to construct reusable components from them. If a customer asks for a component, the developer offers a tender and sells the component.

Component Assembler: Usually, complicated components have to be tailored to match their intended usage. The tasks of a component assembler are to fit pre-built standard components and to integrate them into the system to be built. For smaller systems, this role can be taken over by the component developer or the system architect.

System Architect: The system architect develops a construction plan and selects adequate components, component developers, and component assemblers. Searching for components may be done with the help of specialized component repositories that allow query mechnisms for components organized according to classification schemes, or even with the help of human component brokers. During the construction of the system, the system architect supervises and reviews the technical aspects and monitors the consistency and quality of the results.

Project Coordinator: Project coordinator, as a seperated person, can usually only be found in very large projects. He supervise the whole construction process especially with respect to its schedule and costs. A project coordinator is responsible to the customer for meeting the deadline and the cost limit.

3.5.2 Component-Oriented Development Process

Practical experiences show that neither a pure top-down approach nor a pure bottom-up approach fit very well for software-engineering. Therefore, recent approaches combine these both approaches: Normally one starts to work top-down and changes later to a bottom-up approach. This process is also suitable for component-oriented development which we will motivate in this section.

Top-Down Development

In the top-down approach, development is mainly requirements-driven. First, the customer requirements are analyzed and mapped onto appropriate high-level design components. Then, these high-level components are stepwisely and hierachicaly refined until they can be implemented or until suitable existing components are found (cf. Section 3.2.2).

This pure top-down approach involves some drawbacks: First, refinement is made with no respect to existing components which makes it a matter of luck to result in matching components. Even worse: Although the top-down approach is based on the analysis of customer requirements, it regularly fails to fulfill the existing requirements, if the client initially does not know them or can not state them properly. Last, pure top-down development leads to systems that are very brittle with respect to changing customer requests because the whole system architecture is adjusted to the initially known set of requirements.

Bottom-Up Development

In the bottom-up approach, development builds upon reusable existing components: They are iteratively composed and agglomerated into higher-level components, until, finally, there exists a top-level component fulfilling the requirements for the intended software system.

However, in most cases a pure bottom-up approach is hardly possible because it does not draw into account the user requirements early enough—although the resulting system may be built from reusable components according to a standard architecture, there is no guarantee that it corresponds with the customer's wishes.

To overcome the deficiencies of both extremes, we introduce a new approach. On the one hand, it combines top-down and bottom-up development as equivalent parts complementing each other. On the other hand, it supports new activities pertaining to component reuse.

Figure 4 shows in an idealized and simplified form our additions to the phases of a typical waterfall model—analysis, design, and implementation—to close the gap between top-down and bottom-up development.

While *requirements analysis*, resulting in a business-oriented model, is typically a top-down activity, *system implementation* is usually bottom-up. Starting from a couple of classes, modules and later on components are built.

In the *design* process, top-down and bottom-up development coexist in form of *high-level design* and *low-level design*. Usually, the designer permanently switches between both to reduce the gap between them, resulting in a complete design.

The grey two-way arrows in Figure 4 represent the switching between the typical phases—analysis, design, and implementation. Normally, documents should exist that serve as interface between two phases, particularly if these phases run iteratively or even in parallel.

In a componentware enviroment, the advantages of prototyping are obvious. Prototyping can be seen as a bottom-up way to identify and verify the requirements of the user or to evalute a key-design. Hence, next to analysis and design, *prototype implementation* is very useful. Although these phases eventually run in parallel and influence each other, there may not necessarily be a well-defined interface document. Figure 4 illustrates this fact by the hollow-headed two-way arrows.

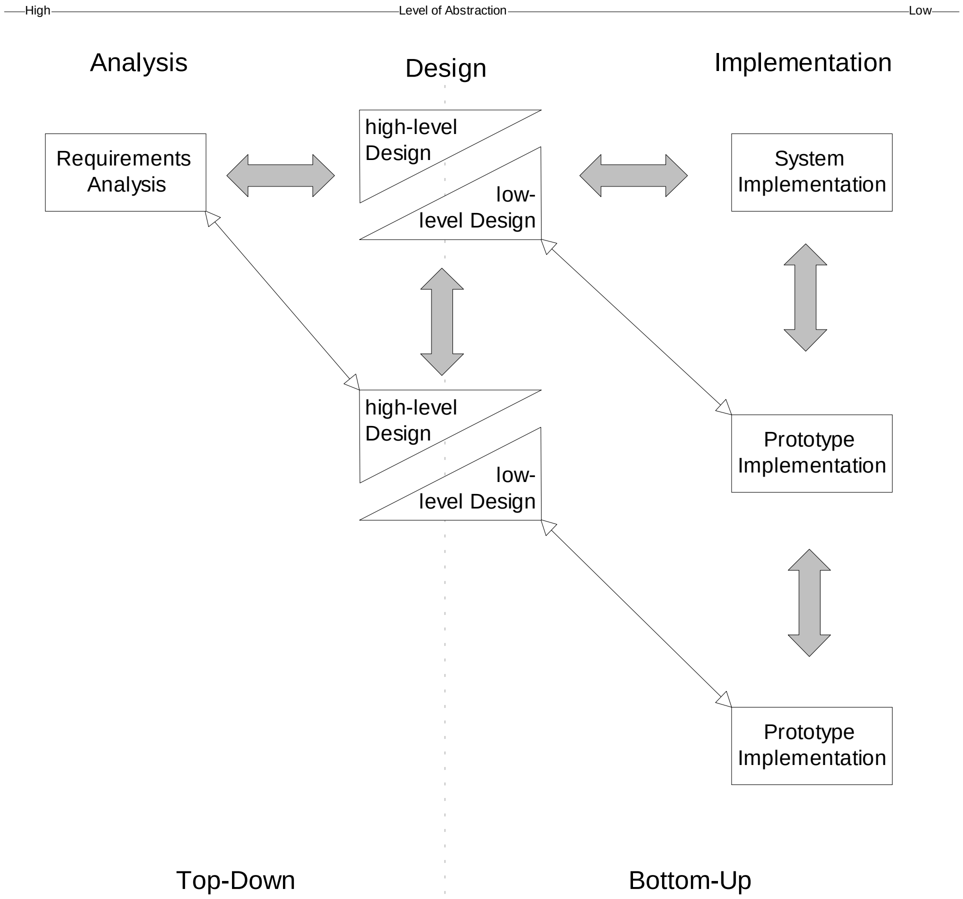

Figure 4: Combining Bottom-Up and Top-Down Development

Besides combining top-down and bottom-up approaches, a component-oriented process should contain activities for finding and reusing existing components during the early phases of analysis and design, and not just in implementation.

Furthermore, not only requirements analysis influence the evaluation of suitable existing components, but also the characteristics of existing components can influence the user requirements. A company may, for example, decide to adopt a standard process as supported by a component and thereby change their origin requirements. Or, explorative prototyping using existing components may help the customer to find and to state requirements. Analogous interdependencies can be found during the design phase, where the selection and evaluation of components influences the system architecture and vice versa.

Figure 5 visualizes the additional activities in a component-oriented process: Each phase not only allows but requires the *search for and evaluation of components*. As said above, this evaluation influences the original phases and vice versa as the hollow-headed two-way arrows indicate. To support prototyping and the final implementation, one has to *order components*. After the *production and shipment of components* by a component vendor, the components will be assembled to a prototype or to the final system.

Obviously, components can also be developed in-house, but even in this case the separation between the different roles mentioned in Section 3.5.1 are essential to support the component-oriented paradigm. The flags in the right upper corner of each activity box in Figure 5 illustrate which roles primarily participate in the corresponding activity.

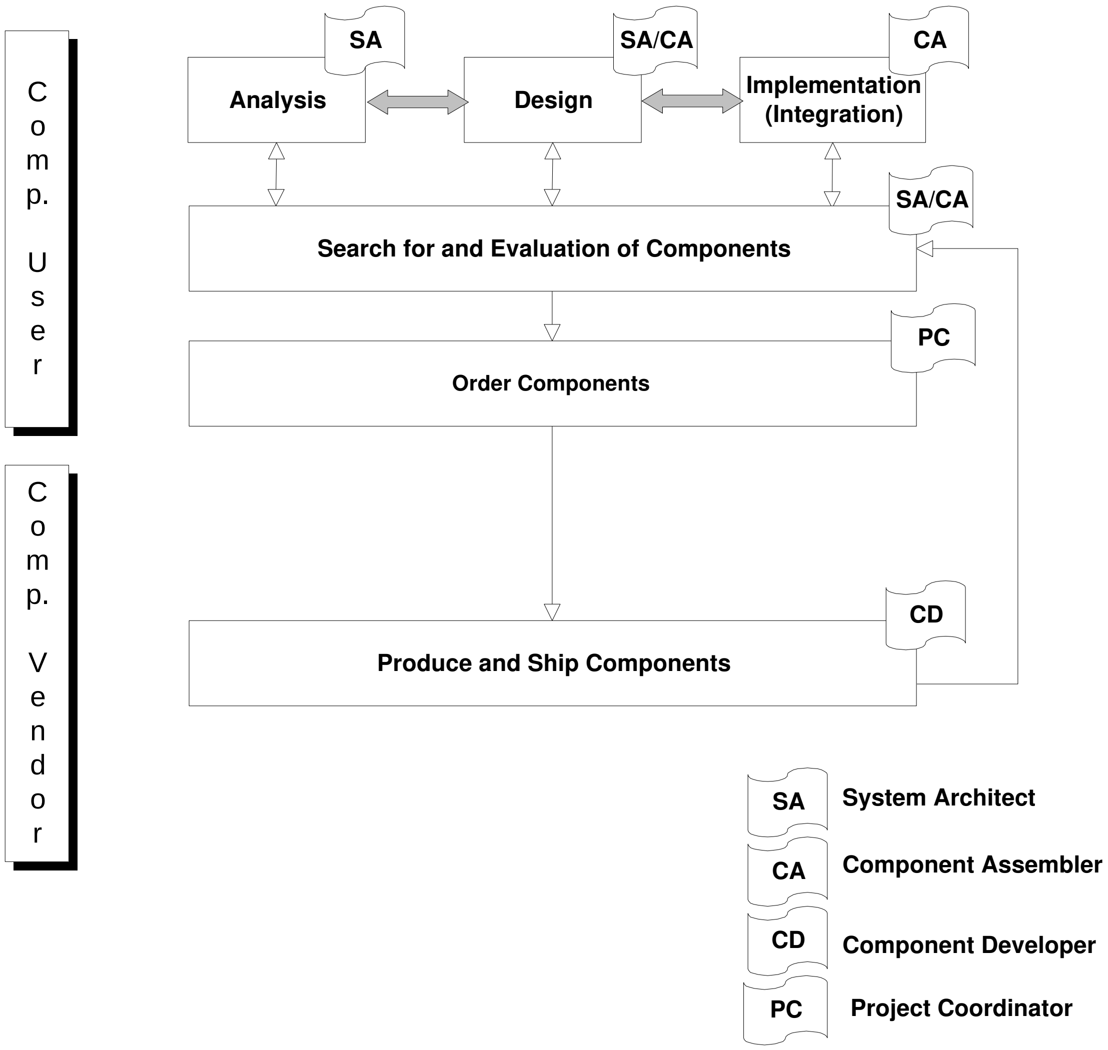

Figure 5: Supporting Component Reuse in a Waterfall Model

3.5.3 The Reengineering Process

The methodical considerations in the last section are admissible only for "pure" component-oriented development. However, our goal in this paper is not to construct a new, component-oriented AutoFocus from scratch, but to redesign the existing, object-oriented tool. Obviously, we need additional techniques for transforming such an existing legacy application to make it suitable for the approach outlined above. While various authors and groups [MLB95, Pro] are concerned with the redesign of existing, procedural legacy systems, we are not aware of approaches for redesigning object-oriented legacy systems.

A big advantage of object-oriented systems is that they are built following a relatively modern software development paradigm allowing the construction of well-structured systems consisting of reusable classes. However, the objects of these classes are very fine-grained building blocks with no hierarchical structure. If high-level components consisting of some cooperating objects exist at all, their interfaces are usually not described explicitly.

Therefore, our approach for redesigning object-oriented systems consists mainly of two activities to identify high-level components within an object-oriented system:

- The functionality of the intended system after the redesign can serve as a clue to find components in a top-down fashion.
- Conversely, the implementation of an existing system can suggest high-level components in a bottom-up fashion.

Whenever one performs the activity *search for and evaluation of components* of Figure 5, these steps have to be done. All other activities can happen according to the process outlined in the previous section.

4 Initial Architecture of AutoFocus

4.1 Overall Architecture

As can be seen in Figure 6, AutoFocus uses a typical two-tier client/server architecture. The repository is document-based, mapping development documents to files in a UNIX file system. Access and version control is provided by the UNIX revision control system RCS [Tic85].

The complete AutoFocus tool, including both the client application and the server application—of course except the underlying RCS—, is implemented in the Java programming language.

The clients, which comprise a set of graphical editors, an application core, and a project browser, are implemented in Java and are thus executable on any platform supporting the Java runtime environment. An arbitrary number of them can connect to the repository to access the documents in the repository.

The network communications mechanism between the repository and the clients is working on top of the standard TCP/IP socket library of Java. Neither RMI (Remote Method Invocation) [SUN97b], the standard mechanism provided by Java 1.1 for implementing distributed applications, nor CORBA for Java [OH97] are currently used in AutoFocus,

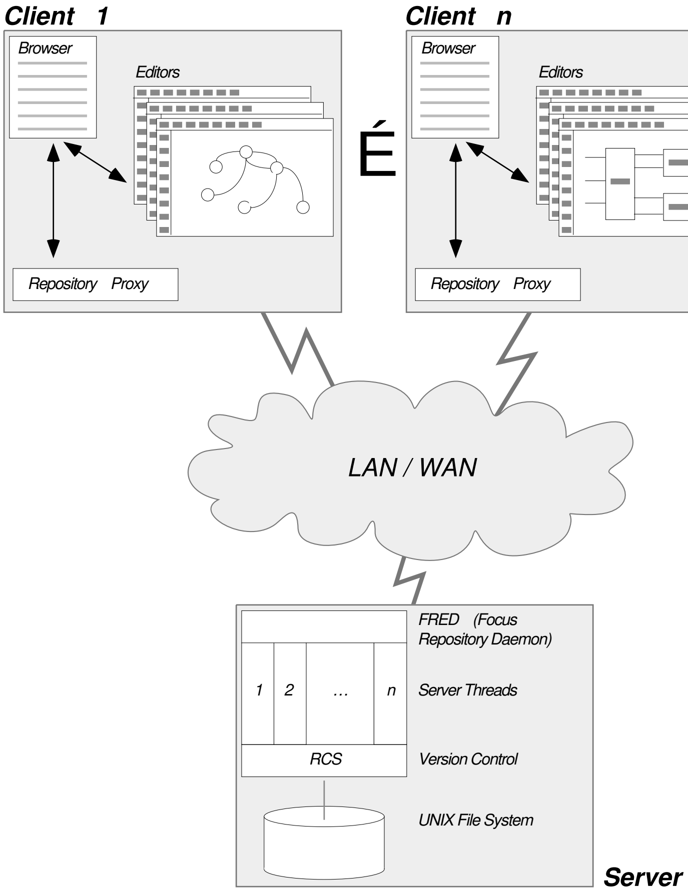

Figure 6: Architectural View of the Initial AutoFocus Implementation

simply due to the reason that they were not available at the time of the first implementation.

Within the server, a dedicated thread is running for each connected client process. Besides handling the access to RCS, the server manages an own list of locked files. This was done as an optimization to prevent external calls to RCS for simple operations like checking for locked documents.

4.2 Problems with the Initial Design

As outlined in the last section, AutoFocus currently uses a document-based approach to modeling systems. Each document is completely independent from all other documents; there is no single, structurally consistent model of the developed system that integrates the different views represented by the documents. Instead, conceptual information and view information is distributed among possibly large number documents. Semantically meaningful cross-references between documents are provided only by equality of certain model element names inside these documents.

The distribution of mixed-up model and view information means that some entities are redundantly represented in different documents. An example are component interface ports in hierarchical system structure diagrams: They appear both in the external blackbox view and in the internal glassbox view of a FOCUS component which are represented by two different documents.

The restriction to this working method has serious drawbacks in practical use:

- The freedom provided to developers can (and will) lead to many inconsistencies making it difficult to integrate the documents into a single, consistent system specification. Especially in a multi-user environment with many developers working concurrently this can be a crucial point. A model-based approach can help here by ensuring certain basic consistency properties.
- Access to the specification is difficult because the information is scattered over many documents. Especially, efficient queries on the repository are not possible in a document-based approach.

Additionally, the current implementation of AUTOFOCUS' document-based approach is very inefficient: It uses RCS-archived files for storing serialized textual representations of the development documents. Complex operations and queries involving many documents must, therefore, load and parse all documents first. This situation could, however, be ameliorated to a certain extent by adding index structures for quicker access. The drawback is here that even more redundancy would be introduced into the repository, making the management of consistency difficult.

For these reasons, we decided to redesign AUTOFOCUS using a model-based repository approach. In this approach, development documents shown in an editor are only views onto a central representation of the specification, the so-called *specification model*. Note that our usage of the notion "model" conflicts with the usual notion in the area of formal methods where a model is not a representation of a specification, but a concrete system fulfilling all axioms and constraints of the specification.

5 Migration Strategy: Redesign Steps

In the following, we will explain the steps we have performed during the component-oriented redesign of AUTOFOCUS. They follow the process outlined in Sections 3.5 and 3.5.3.

5.1 Identification and Transfer of Components

In the case of AUTOFOCUS, the identification of components is not very difficult because the system has been designed according to a well-structured construction plan. So, the component candidates can mainly be derived from the current subsystems of AUTOFOCUS.

5.1.1 Repository

First of all, the whole repository subsystem of AUTOFOCUS represents a candidate for a large component. Such a repository component provides, among others, the following services to client applications:

Persistence Documents can be persistently stored in the repository, available for retrieval at any time. As mentioned before, this is currently done by streaming documents into RCS-controlled, flat files inside a special directory in the file system.

Versioning The repository provides services pertaining to version management, that is, storing the complete development lifecycle of documents, not only the actual state. Internally, versioning is provided by RCS, on top of which the repository server runs. Version numbering is done via RCS version numbers.

Access Control and Locking In a typical multi-developer environment, access conflicts to a development document will arise when two or more developers try to modify the same document simultaneously. AutoFocus' stragtegy to handle such potential conflicts is a simple MROW (Multiple Readers, One Writer) strategy. It is implemented by means of locking mechanisms keeping track of read and write accesses to documents.

Currently, these services are all provided by the repository server of AutoFocus. However, their different, clearly distinguishable functionality suggests a separation into three different components.

Transaction Management and safe storage —a service that should normally be provided by a repository, too—is currently not available in AutoFocus. It should, however, be part of a future repository concept.

An evaluation of the current repository implementation yields that the present code can hardly be reused in the components of a new repository: It depends strongly on RCS and lacks some of the functionality of a "real" repository. We decided, therefore, to drop the current repository altogether and to focus on this part during the redesign in Section 6.

5.1.2 Network Communications

AutoFocus is designed and implemented as a distributed system where clients and server typically run on different, communicating computers. The communication functionality can roughly be divided into two functional areas:

- Document must be transferred from the repository to the clients and vice versa, and
- messages between the server and the clients must be sent that contain, for example, status information about locking and unlocking of documents by other clients.

Again, one can see the whole networking functionality of AutoFocus as a large networking and communication component, perhaps with two sub-components for content delivery and signalling, respectively.

In the current implementation of AutoFocus, the communication functionality is programmed, as already mentioned above, using the standard Java socket library. Today, more modern mechanisms are available that can be used more or less transparently.

We therefore decided to drop the old communication subsystem of AutoFocus altogether and will replace it by RMI or CORBA for Java.

5.1.3 Client Application Core

On the client side of AutoFocus, a so-called *coordinator*, implemented as a singleton object, is responsible for coordinating all actions within the client. It also keeps track of all local status information, like, for instance, which documents are currently open or edited. The coordinator is a very natural candidate for a component, perhaps with subcomponents for managing open viewers or communication with the repository.

The coordinator of the initial version of AutoFocus was implemented by very tangled and ad-hoc written code without clear interfaces. Therefore, we have decided to drop the old code and to reengineer this component.

5.1.4 Project Browser

The project browser is the main user interface for navigating through the contents of the repository. It can quite naturally be considered a component itself.

Considering that the information displayed in the browser is mainly the information available in the client's application core, the browser is basically a view onto the core. The client application core can be regarded as the corresponding model in a model-view relationship.

As the browser depends strongly on the client application core and its code was also very tangled, we decided to reengineer this component also, using a ready-made treeview component programmed by one of our students.

5.1.5 Editors and Viewers

Candidates for other very important view components are the editors and viewers of AutoFocus. They are used to visualize and edit the data in the repository. One can see them as medium-sized components consisting of several subcomponents. For instance, all kinds of dialogs that provide access to properties of specification elements in documents can be identified as an additional category of visual view components in AutoFocus.

Although the editors and viewers of AutoFocus have many deficiencies with respect to their user interface, they offer sufficient functionality for the basic editing tasks. We decided, therefore, to keep these components and to improve the user interface as well as the components functionality.

5.1.6 Documents

Finally, the fine-grained document objects together with their subordinated objects—like states and transitions in the case of a state transition diagram—can also be considered as candidates for components.

We did in fact keep the documents during the redesign, but gave them a new role as the views of the new, model-view-based repository architecture (see Sections 5.2, 6.1.2, and 6.1.3).

5.2 Addition of Model Layer

As explained in Section 4.2, the initial version of AutoFocus did not know the separation of model elements and their corresponding views. Instead, documents played both roles: They contained the model information and were also responsible for the presentation of this information on the screen.

To achieve a seamless migration to the new, component-oriented and model-based design as introduced in Section 6, we decided to add a model layer to AutoFocus. Our strategy here is to generate the model layer on demand from the current document views. That will, of course, only result in a unique model if the documents fulfill certain consistency criteria (cf. next section).

The intermediate step of generating the model from the views was mainly done because it saved us a lot of time during the preparation of the practical software engineering course in which the simulation component was developed (cf. next section). Later on, we will revert the dependency between models and views: The views will then be based on and generated from their model elements, as explained in Section 6.1.3.

5.3 Addition of Simulation Functionality

The last addition to AutoFocus was a simulation environment that allows to trace the execution of a specified system [Sim]. Its main component is the simulation generator which takes a specification and generates an executable simulation which itself can be considered a component. This executable corresponds with a multimedia component to visualize the impacts to the outside world (e.g. controlling of street lights, etc.)

5.4 Addition of Consistency Checking

Prototypical support for consistency checks is also already integrated into AutoFocus. The work in that area was focused on the design of a language for the expression of consistency checks, for which an interpreter was built. The connection to the model elements is yet very ad-hoc and does not follow the generic architecture described in Section 6.8. However, we think that the existing interpreter can be used as a component in the new design.

6 New, Component-Based Architecture

The next sections cover the new, component-based architecture of AutoFocus, with the focus set to the repository functionality lying at the heart of the tool. We first present the overall architecture of the system and describe the design principles and structuring schemas for the involved components. Section 6.2 covers base concepts like the representation of model and view elements, and Sections 6.5 until 6.8 cover the main features of the system—like versioning, persistence, and locking—and their realization. The following, last subsection describes how the resulting business-oriented design can be mapped to a distributed hardware.

6.1 Overall Architecture

As said in Section 2, the main goal of the redesign is to make AutoFocus a flexible toolkit for experimentation with different meta-models, working methods, and new CASE tool concepts in general. Therefore, extensibility and flexibility with respect to the following issues is required:

- Meta-modelers and methodologists are offered a rich and comfortable framework with base mechanisms for all kinds of meta-models and working methods. New meta-models and description techniques can be easily integrated.
- The tool can be easily extended with new mechanisms and functionality, and the architecture is modular enough to allow changing, modifying, and re-combining the subsystems of the implementation.

We tackle these requirements by structuring the system according to three dimensions, as shown in Figure 7:

1. We distinguish the representation of the meta-model from the part of the system responsible for the mechanisms (see Section 6.1.1),
2. we group all interfaces and classes into three layers as will be shown in Section 6.1.2, and
3. we use a modern Model-View architecture distinguishing model components from their corresponding views (see Section 6.1.3).

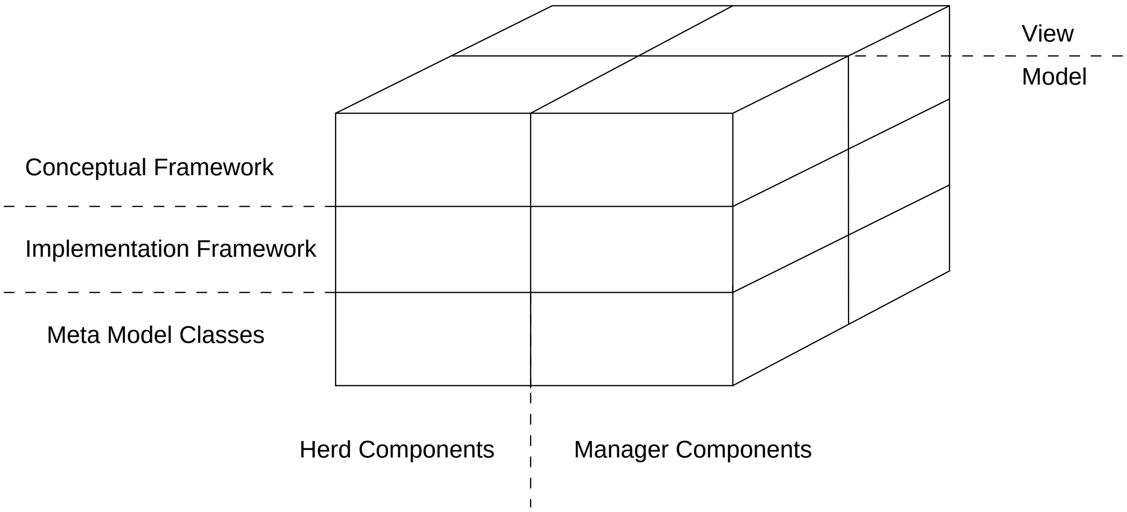

Figure 7: Design Dimensions of the New AutoFocus Architecture

6.1.1 System Mechanisms and Herd Components

The herd component part of the system consists mainly of interfaces and classes for modeling and representing the description techniques of various modeling languages. At runtime, each modeling construct of type ModelElement—like, for example, a graphically represented state or transition— is a herd component managed by the system's managers.

The mechanism part consists of various manager components that provide the more advanced functionality of the system. Managers are typically singleton components—there is only one manager of a kind in a system. An example for a manager is the VersionManager. It is responsible, among others, for creating new version numbers and for returning the actual version of a certain ModelElement.

New managers can be added easily if they are orthogonal to existing mechanisms. Otherwise, the resulting interactions and overall mechanisms must of course be carefully designed. An example for a non-trivial dependence between two mechanisms can be found between versioning and locking: One can not create a new version branch from a document locked exclusively by someone else, so versioning depends on locking and we need to coordinate their managers following a global strategy.

6.1.2 Layered Framework

We do not use different languages or representations for the meta-model and the meta-meta-model in our approach. Instead, both models are expressed in a single, layered framework consisting of Java interfaces and classes. This allows a seamless transfer of common classes used for all or many meta-models into the framework by abstracting out common concepts of meta-models.

Another goal was to find an open solution where the framework is independent from a special implementation and can be accessed from other languages remotely. This led us to an architecture with three layers:

Conceptual Framework This layer contains only Java interfaces. It models the base concepts and can be considered as a meta-meta-model because all concrete meta-model and manager classes must implement its interfaces. The conceptual framework contains only abstract concepts independent from a concrete implementation. Besides that, using only interfaces makes it easy to allow remote access via RMI or CORBA remote interfaces [SUN97b, OH97].
Example interfaces contained in the conceptual framework are ModelElement, from which all meta-model classes must be derived, Lockable, which must be implemented by all meta-model classes that can be locked for access by a user, and LockManager.

Implementation Framework This layer consists of abstract Java classes providing standard implementations for some of the abstract concepts of the conceptual framework. Usually, new meta-model classes and new managers will build on the standard implementations of this layer.
An example class contained in the implementation framework is Document offering the functionality needed by "document-level" modeling constructs which can be locked and edited by users as a whole and may have multiple versions.

Meta-Model Classes This layer captures the concepts of a certain modeling language by means of ordinary Java classes and provides concrete implementations of corresponding managers for the system's mechanisms.
Example classes in the context of state transition diagrams are the meta-model classes TransitionModel or StateModel.

Although structuring the framework into three layers provides for easy extendability, evolution of the framework is in general no easy task: If the basic concepts in the conceptual framework change, not only all implementations in the implementation framework and the meta-model classes must be adapted, but also all existing, concerned meta-model instances.

6.1.3 Model-View Architecture

Another structuring dimension mirrors our decision to base AutoFocus on a model-view architecture. It is a slightly adapted variant of the well-known model-view-controller architecture, but with the controller integrated in the view (many modern architectures follow this approach [Mic96, Java]). Views exist for most of the meta-model elements and manager components of the system. There may of course be more than one view for a single model.

6.2 Base Concepts: Models and Views

All meta-model elements must be derived from the interface ModelElement. Model elements are observed by a number of view elements derived from the interface ViewElement. The views observe their corresponding model following the so-called *Observer* pattern [GHJV95].

Examples for concrete view elements are graphical representations of document-level model elements and basic graphical elements like lines or ellipses that correspond to model elements like transitions or states.

Analogous base classes can be found in the mechanism part: All managers must be derived from the interface Manager, while the corresponding views must be derived from the interface Viewer.

An example for a concrete manager and its pertaining viewer is the lock manager which could be visualized by a window showing a list of locked document-level model elements together with the locking users.

Components and interfaces:

ModelElement: Interface for all herd model elements, must be implemented by all meta-model elements.

ViewElement: Interface for all views onto model elements. A view can register itself at its corresponding model element following the *Observer* pattern [GHJV95].

Manager: Interface for all manager components.

Viewer: Interface for all views onto manager components. Registration of a viewer at its corresponding manager also follows the *Observer* pattern.

Figure 8 shows the interfaces for the model-view-mechanism. Interfaces are shown as boxes with rounded corners. Note that, although Observer can be taken from Java's standard framework, Observable can not be taken from there, because it is a class, and we want to have only interfaces in the conceptual framework.

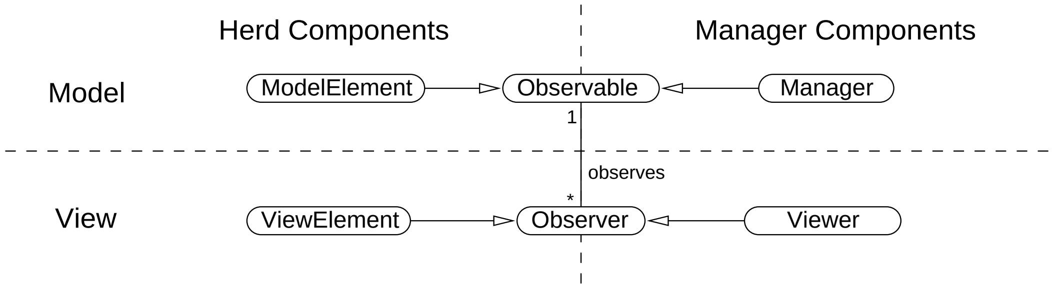

Figure 8: Model-View Architecture in the Conceptual Framework

6.3 Entities, Relationships, and Navigation

While the fixed connection structures of the singleton manager components are easy to handle, this is not true for the changing graph structure of meta-model instances. Therefore, we have to provide adequate support for navigation in this area. We do this by introducing two sub-interfaces from ModelElement, namely Entity and Relationship, with the usual semantics: Entities can be connected via bidirectionally navigatable relationships.

Connection structures like this are usually modeled by introducing methods like Entity::getRelationships, Relationship::getLeft, and Relationship::getRight. Special relationships, like, for example, Aggregation would be derived from Relationship, and they would connect special entities of types Whole and Part, with methods like Aggregation::getWhole, Aggregation::getPart, Whole::getAggregationsToParts, and Part::getAggregationsToWhole.

However, this has some disadvantages:

- The functionality for navigation is not contained in a single interface, but scattered over some interfaces (e.g. over Aggregation, Whole, and Part). Changes in the implementation of a relationship usually require changes to all implementations of these classes.
- If one wants to extend a meta-model with specialized functionality for navigation, like, for example, a method returning only Relationship instances satisfying certain criteria, new entity interfaces with additional methods must be introduced. This leads to a plethora of different entity concepts that differ only with respect to navigation.
- Entities participating in two different relationships of the same type can not be modeled adequately, as can be seen in the following example: The type Boiler is aggregated along two different dimensions. According to the spatial aggregation hierarchy, a boiler is part of a certain room of a building, and according to the functional aggregation hierarchy, it is part of the whole central heating. However, Boiler can only implement Part once, forcing one to introduce two new, specialized Aggregation relationships like, for example, SpatialAggregation and FunctionalAggregation, along with the types SpatialWhole, SpatialPart, FunctionalWhole, and FunctionalPart. Again, this leads to a plethora of different meta-model concepts.
- Adding a new relationship to an archived entity supposed to be immutable (cf. Section 6.4) would change the result of its Entity::getRelationships method (analogously for the method Whole::getAggregationsToParts and similar methods).

Because of these disadvantages we decouple the information about connected relationships from entities and strip entity types of any functionality for navigation. Instead, we use so-called *navigation manager* components to return the relationships connected to a certain entity. This leads to the following pattern of interfaces for the Aggregation relationship:

Aggregation represents a single, bidirectional aggregation link. It is derived from the generic interface Relationship and contains the methods getWhole and getPart for accessing the two partners of the aggregation. Besides that, relationship interfaces may also contain special constraint and check methods and even complex, application-dependent functionality introduced by the meta-modeler.

Whole/Part are bare interfaces without any methods. They provide type-safe access to the connected entities.

AggregationManager is a specialized navigation manager derived from the generic interface RelationshipManager. It provides at least the methods getAggregationsTo Parts(Whole) and getAggregationsToWhole(Part) for returning all connected aggregations for a certain entity. More advanced functionality, for example, methods like getPartsFor(Whole), getTransitivePartsFor(Whole), or getRootFor(Part) are also possible. Missing functionality can be easily added by deriving new, specialized navigation managers from AggregationManager.
Note that different navigation manager instances for different aggregations (like for the spatial and functional hierarchies in the example above) can coexist without problems.

The resulting architecture can be seen in Figure 9. It omits only the model/view architecture dimension and shows only the model part. Interfaces are shown as boxes with rounded corners, while abstract classes are shown as slanted boxes, following the notation in [Fla96]. We do not provide a special navigation architecture for the view part because it can be built using standard GUI implementation techniques outside the scope of this paper.

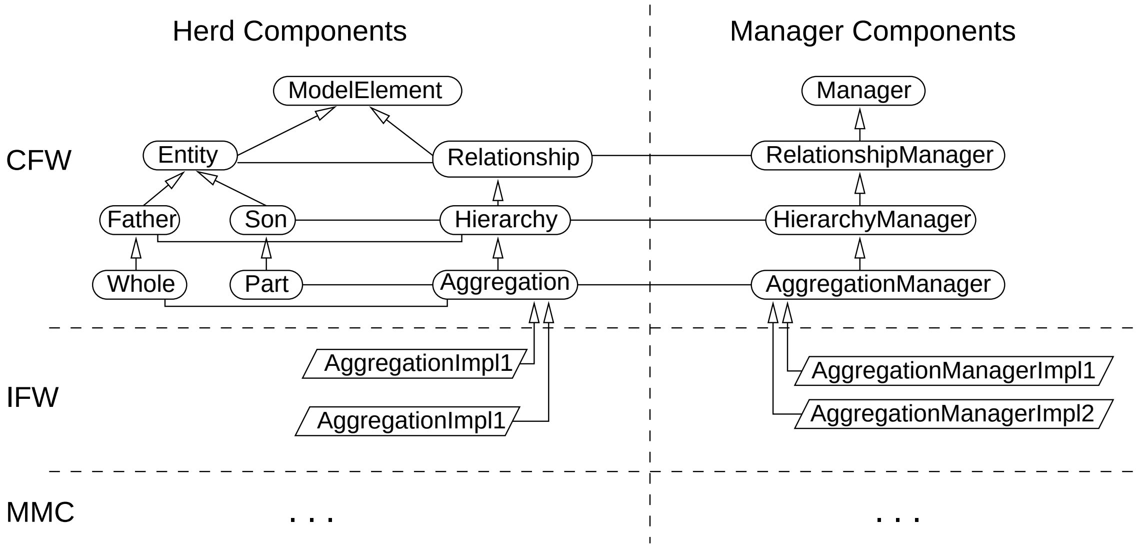

Figure 9: Relationship Architecture for Model Part

Apart from Aggregation, we provide also some other specialized relationships we think are useful in every meta-model: Hierarchy models tree-like structures (for example Aggregation relationships), Refinement models simple implementation relationships, and Compatibility models mutual exchangability of two meta-model elements.

Hierarchy/Father/Son: Hierarchy is derived from Relationship and models directed, hierarchical relationships. It contains functionality for navigating to the connected Father and Son entities.

Aggregation/Whole/Part: Aggregation is derived from Hierarchy (accordingly, Whole and Part are derived from Father and Son), and handled analogously.

Refinement/Abstract/Concrete: Refinement is derived from Relationship and handled analogously to Hierarchy/Father/Son, only that refinement is a m:n-relationship. Complex refinements where proofs or other complex data are involved can be modeled by subclasses.

Compatibility/Compatible: Compatibility is derived from Relationship and handled analogously.

Each of these specialized relationships is managed by its special navigation manager.

6.4 Persistence and Archiving

Our archiving concept comprises two separate areas, namely, the *archive* with old, immutable versions of model elements, and the shared *workspace* with the actual, "live" model elements that are worked on by one or more users. View elements are also stored persistently because one has to keep, for example, the geometry information pertaining to each view. New versions of model elements are created only on user archiving requests, not on every user edit action.

Relationships between entities in the archive can not be changed or removed. However, this is in principle possible for relationships between workspace model elements and archived model elements, as well as for relationships between workspace model elements and workspace model elements. The conditions under which new relationships can be introduced depend on each meta-model; we have, therefore, decided not to provide additional support for them in our framework architecture.

We have not introduced a delta mechanism for storage space efficiency into our design because we think that would be a premature optimization at this time: All versions of model elements are considered as full, persistent objects. Persistence is handled transparently by a persistent object system like PJama [Lab] or the CORBA persistence service [COR, OHE97]. We have not yet decided which persistence mechanism to use, as this requires extensive evaluation.

However, we have provided a specialized manager component responsible for managing all archived model and view elements.

Components and interfaces:

ArchiveManager: This manager keeps write locks (cf. Section 6.6) to model elements and thus prevents users from changing them. It is also responsible for answering global queries to the archive, like, for example "Return all actual model element versions as of February, 1st." in cooperation with the version manager (cf. next section).

6.5 Versioning of Model Elements

Our version concept allows sequential and branching versions for each versioned entity, as visualized in figure 10. Whenever two branches are folded into one version, their consistency must be checked by some means to prevent inconsistencies in the resulting version.

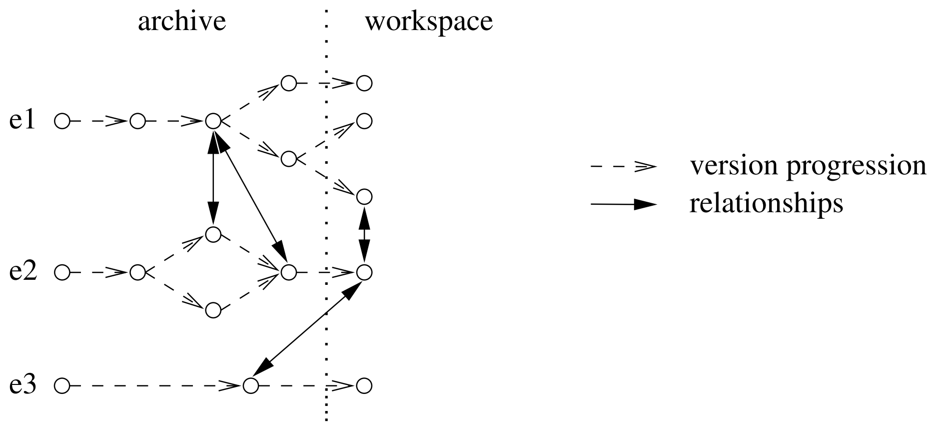

Figure 10: Versioning Concept

Whenever a user wants to archive the current state of an active workspace model element, new sequential versions of all transitively reachable, not yet archived model elements in the workspace are generated, too. The elements in the workspace are not influenced by such an action, however. Figure 11 shows this behavior; in this example, the black and the grey model elements in the workspace were modified compared to their last archive versions, whereas the white model element was not changed (the relationships themselves, visualized by the two-way solid arrows, were also not changed). An archiving request for the black model element leads to the situation on the right, where the modified grey element is archived, too.

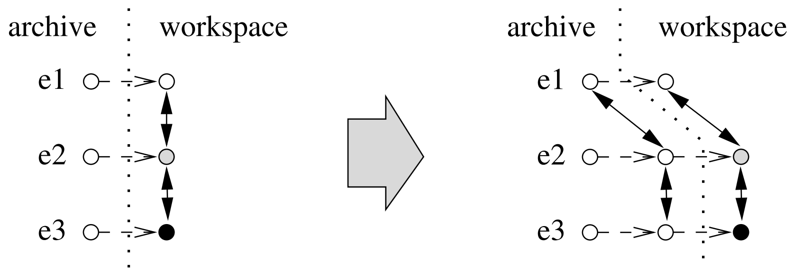

Figure 11: Archiving of a New Model Element Version

Whenever a user wants to create a new variant—that is, a branch—of an existing workspace model element, a new archived version of the model element must be created. Branches of archived versions can be created without creating a new version. Figure 12 shows both possibilities; the black model elements are branched.

We consider a the choice of a concrete version numbering scheme a presentation option not relevant for system design. A possibility is, for example, to use the version numbering scheme used by RCS [Tic85].

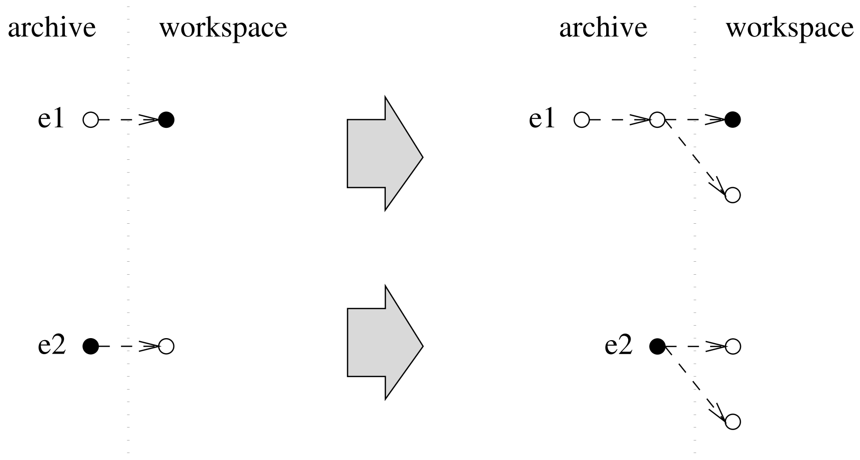

Figure 12: Branching of Model Element Versions

Note that we did not introduce a *snapshot* concept for whole systems: If a user wants to store the current state of all model elements together, the meta-model must provide a system model element referencing all needed model elements in the model.

Another point we want to stress is that usually not all model elements are versioned. However, all non-versioned element must be encapsulated in and handled by versioned model elements, and there may be no relationships to them from model elements outside of this encapsulating model element. Additionally, relationships between versioned model elements are usually also versioned, especially if they carry application information. This leads to a two-leveled structure with *document-level* model elements that can be versioned and locked by users (cf. next section), and *low-level* model elements contained in the document-level elements.

View elements are, like low-level elements, not versioned themselves; their version follows the version of their viewed model element.

Components and interfaces:

Versioned Interface for herd model elements, must be implemented by all versioned model elements. Contains functionality for storing and accessing version information of a single model element, and for navigation to the predecessors and successors in the version history.

VersionManager Manager component, responsible for implementing a version numbering scheme, for realizing complex queries on the version history (e.g. "Return all actual workspace model element versions for a certain model element."), and for performing complex changes on the version structure (e.g. archiving all model elements that are reachable from a certain model element in the workspace).

6.6 Locking and Checkin/Checkout

As explained in Section 2, on of the main goals of the AutoFocus redesign is to allow experimentation with different working methods. We distinguish the following main variants:

Shared Whiteboard Some users may simultaneously have the right to edit a single model element, and they see all modifications in real-time. If conflicts arise, they are resolved according to some strategy. If, for example, two users attempt to drag a graphically represented element to different locations on the screen at the same time, only the action started first will succeed.

Exclusive Whiteboard At each time, only one user has the right to modify a single model element, but all users see the modifications in real-time. Elements owned by a user must be accordingly marked to indicate that they can not be worked on by other users willing to edit them.

Checkin/Checkout At each time, only one user has the right to edit a single model element, and no other user can see his or her modifications. Elements that are checked out by a user must be accordingly marked to indicate that they can neither be edited nor seen by other users.

We define three kinds of access modes that correspond to these working methods:

Read Access corresponds to the checkin/checkout working method. It indicates that a user reads a certain model element ('reading' means in this context that the user has opened an editor with a representation of the model element). Multiple simultaneous users with read access to a single element are possible.

Write Access corresponds to the exclusive whiteboard working method. A user with write access has the exclusive right to modify a model element. Although only one simultaneous user with write access to a single element is possible , there may additionally exist many users with read access.

Exclusive Access corresponds to the checkin/checkout working method. A user with exclusive access has the right to modify and read a model element. As the name indicates, there must not be other users with access to a model element that is accessed by a user with exclusive access.

Access modes are not a technical concept, but must be visualized to the user. A possible, straigtforward implementation will use the technical concept of locking. In the following, we present a simple concept for lock management, where read, write, and exclusive access is realized by read, write, and exclusive locks, respectively. The final implementation may not necessarily be based on this locking concept, but may resort to a built-in locking mechanism of the used persistence platform (cf. Section 6.4).

As with versioning, not all model elements must be necessarily lockable. However, all non-lockable low-level elements must be encapsulated in and handled by lockable document-level model elements, and there should be no relationships to them from model elements outside of this encapsulating document-level element.

The lock granularity must be defined by the meta-modeler. Apart from the implicit propagation to encapsulated low-level elements of a document-level element, the meta-modeler can choose to propagate the lock to other model elements. An example arises in the context of AutoFocus' hierarchical state diagrams: Here, the meta-modeler could implement a locking strategy where a lock set on a sinlge state locks also all subordinated sub-states and their transitions recursively. Locks can not be set on single attributes of model elements; if a meta-modeler wants to have very fine-grained locking control, he or she has to choose adequately fine-grained meta-modeling concepts.

As with versioning, view elements are not locked themselves; their locks follow the locks of their viewed model element.

Locks can be used to implement the separation between archive and workspace elements by allowing the ArchiveManager to achieve write locks to the archived model element versions that it never returns.

Components and interfaces:

Lockable: Interface for herd model elements, must be implemented by all lockable model elements. Contains functionality for setting and accessing information about each of the locks of a single model element.

Lock: Interface for the three kinds of locks. Allows access to information about

- the kind of the lock (read/write/exclusive),
- the user who owns the lock, and
- the time at which the lock was created.

Lockmanager: Manager component, responsible for complex queries concerning locks (e.g. "Which state transition documents has user Quargl open for reading?"), and for locking whole hierarchies or transitive closures of model elements with the help of navigation managers.

6.7 Transactions and Undo/Redo

Users can control a CASE tool in various ways: There will usually be interaction via menus, dialogs, mouse actions in graphical editors, or even via automation facilities like scripting languages (cf. next section). All of these interactions can be captured using the concept of *commands* as introduced in [GHJV95]. Apart from providing a single, powerful abstraction for all kinds of user interaction, commands offer support for undoable and redoable actions.

In the context of a multi-user tool, where many users access a set of shared resources, we also need a facility for coordinating the user commands. This can be done most easily by encapsulating each critical command sequence in a transaction. The choice of transaction granularity and generally the implementation of the single command transactions is not part of our architecture; it must be designed according to the intended working method by meta-modelers and methodologists extending our framework. The shared whiteboard working method would, for example, lead to very fine-grained transactions on the level of single mouse actions.

We do not intend to program support for transactions by ourselves because components for this purpose are already available: The persistence mechanism PJama [Lab] or CORBA's transaction service [COR, OHE97] are possible alternatives here. We have not yet decided which alternative to choose, as this requires extensive evaluation.

Components and interfaces:

Command: Interface for all kinds of actions that can be performed by the users. Contains functionality for execution, undo and redo.

Transaction: Derived from Interface for commands accessing shared resources. Contains functionality for commit and rollback, as required by the underlying transaction mechanism.

6.8 Access for Scripting Languages

A CASE tool like AutoFocus should be very configurable. One possibility to improve configurability is support for various scripting languages, like, for example, a simple language based on predicate logic for formulating consistency conditions [HSE97] on models.

Scripting language interpreters need access to certain methods of model elements, but must not access other methods. An example is the meta-model type State. Its method isFinalState should be callable in a consistency check, whereas the method deleteObservers (disconnecting the views from the model element) should not be callable. Therefore, we need a generic mechanism for fine-grained access control to model elements methods.

We provide the interface Scriptable for this purpose: It contains methods for characterizing some of the methods of a meta-model interface as callable from a scripting mechanism. The actual invocation of callable meta-model element methods can be done via Java's reflection facility [SUN97a] or CORBA's dynamic invocation facility [OH97].

Components and interfaces:

Scriptable: This interface contains at least the method mayExecute(String) that checks whether a certain method of a model element may be dynamically invocated by a scripting language interpreter.

ScriptManager: This interface contains functionality for the dynamic invocation of a model element's method. If the method must not be called by the corresponding scripting language, the script manager returns an exception.

ScriptInterpreter: This interface contains functionality for parsing and interpreting a script language. For executing methods on model elements, it relies on the corresponding script manager.

Specialized, derived interfaces must be provided for specialized scripting languages. Examples for the language of consistency checks are the interfaces Checkable, CheckManager, and CheckInterpreter.

6.9 Distribution Design

Distribution design is concerned with the partitioning of the data and functionality of a system on a network of physically or logically distributed computation nodes.

Our intended, typical target architecture is a powerful central server with some Java-capable clients. For this configuration, the easiest and most flexible solution is to centralize as much data and functionality on the server, leaving only the graphical user interface data and functionality on the distributed clients. This has the following reasons:

- Specification model information can not easily be assigned to a single client. Especially in the shared whiteboard working method, many users must access the same elements. Centralizing all information on the server keeps the communication patterns simple. It also avoids the necessarity for object migration of model elements from one node to another.
- The single elements of the specification model are strongly interconnected. If some of them would live on the clients, this would result in many remote references and

in complicated reference structures between the clients. If all data is held on the server, all references between model elements can be implemented by server-local Java references.

- Queries and global operations that need the whole specification model can be executed efficiently on a central server.
- Data security and access control is easy with a central server.

While centralizing the specification model on the server makes no difficulties, there is a problem with the view components: As we allow working methods—for example, in the shared whiteboard approach—where many users have access to the same view information, not only the model elements, but also the view components live on the server. This implies that there exist other, "second-level" views on the client that draw the screen representation and react on user actions. In this architecture, first-level server views observe their corresponding model elements, while the second-level client views observe their corresponding first-level server view.

The main disadvantage of the proposed distribution architecture is that the load on the network connections between the clients and the server is high: Each user interaction usually requires some calls to server components. However, we think that a first implementation should not contain premature optimizations that would complicate the overall architecture. If they turn out to be necessary in certain areas, such optimizations can be added later on.

A possible extension would be to add other servers for special tasks to this architecture. This makes sense, for example, for functionality like automated proof support via model checkers or theorem provers: These tools are usually large, not very portable legacy components that need fast computing servers to perform adequately. Wrapping their functionality in a remote interface would allow to connect them be used from AutoFocus or other tools.

The technical realization of the distributed communication can be done via a distributed programming mechanism. We have evaluated RMI [SUN97b] for that purpose [BRS] extensively and will evaluate also CORBA for Java [OH97]: Both approaches seem suitable for inclusion as communication management components into our system.

7 Conclusion

In this paper, we have outlined a methodology for component-oriented software development. The methodology was in turn applied to a complex development problem, namely, the redesign of the distributed multi-user CASE tool AutoFocus.

Currently, only parts of the redesign are realized:

- The client-side browser and application core were reengineered and connected to the old, RCS-based repository.
- The functionality for simulation was added in a practical software engineering course during summer terms 1997 [Sim]. As part of the preparation of this course, the specification model for the existing development techniques was added. However, it is not yet connected fully to the views, as explained in Section 5.2.

- A prototype of the consistency checking mechanism was implemented [Ein97].

Although the current version of AutoFocus provides a working environment that is sufficient for small projects, most of the work for the redesign remains yet to do. This pertains not only to the actual coding, but also to the elaboration of the proposed design: Existing base techniques and components—especially for transaction management, persistence, and distributed communication—must be evaluated, interface signatures must be defined, and a prototype must be built to show the viability of the approach.

Similar considerations apply to the componentware concepts and the methodology introduced in Section 3: They must be refined and developed further, for example with respect to existing, technical componentware approaches like Java Beans [Javb], CORBA [COR], or COM/DCOM [Bro95], and adequate process models, description techniques, formalisms, management techniques, and ultimately tools must be found and constructed.

Acknowledgements

We thank Bernd Deifel, Sascha Molterer, and Alexander Vilbig for interesting discussions and comments on earlier versions of this report.

References

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber. The design of distributed systems - an introduction to FOCUS. Technical Report TUM-I9203, Technische Universität München, Institut für Informatik, Januar 1992.

[Ber97] Klaus Bergner. *Spezifikation großer Objektgeflechte mit Komponentendiagrammen.* CS Press, 1997.

[BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language, Version 1.0.* Rational Software Corporation, URL: `http:/www.rational.com`, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951 (USA), 1997.

[Bro95] Kraig Brockschmidt. *Inside OLE2.* Microsoft Press, 2nd edition, 1995.

[BRS] Klaus Bergner, Andreas Rausch, and Marc Sihling. Casting an abstract design into the framework of Java RMI.

[COR] Object Management Group CORBA. OMG website, `http://www.omg.org`.

[DPB96] Wolfgang Keller Dr. Peter Brössler. Von monolithen zu komponenten - wege zu objektorientierten software-architekturen. In Heinrich C. Mayr, editor, *Beherrschung von Informationssystemen : Tagungsband der Informatik '96*, Österreichische Computer-Gesellschaft : Schriftenreihe. R.Oldenbourg , Wien, 1996.

[Ein97] G. Einert. Dokumentübergreifende Konsistenzprüfungen für das Werkzeug AutoFocus, 1997. Fortgeschrittenenpraktikum.

[Fla96] D. Flanagan. *Java 1.1 in a Nutshell.* O'Reilly & Associates, Inc., 2 edition, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Hin97] Dietrich Hinz. *Die neue HOAI.* Forum Verlag Herkert GmbH, Merching, 1997.

[HSE97] F. Huber, B. Schätz, and G. Einert. Consistent graphical specification of distributed systems. In *FME'97*, Lecture Notes in Computer Science. Springer, 1997.

[HSS96] Franz Huber, Bernhard Schätz, and Katharina Spies. AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme . In Ulrich Herzog Holger Hermanns, editor, *Formale Beschreibungstechniken für verteilte Systeme*, pages 165–174. Universität Erlangen-Nürnberg, 1996. Erschienen in: Arbeitsbereichte des Insituts für mathematische Maschinen und Datenverarbeitung, Bd.29, Nr. 9.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AutoFocus - A Tool for Distributed Systems Specification . In Joachim Parrow Bengt Jonsson, editor, *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470. LNCS 1135, Springer Verlag, 1996.

[IT93] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, September 1993.

[Java] JavaSoft. Java Foundation Classes website, `http://java.sun.com/products/jfc/`.

[Javb] JavaSoft. JavaBeans website, `http://java.sun.com/beans/`.

[Lab] Sun Microsystems Laboratories. PJama Release 0.3.4 (for JDK 1.1.4), `http://www.sunlabs.com/research/forest/opj.main.html`.

[Mic96] Microsoft Corporation, Redmont, WA (USA). *Visual C++ 4.0 and Microsoft Foundation Class Library Manuals*, 1996.

[MLB95] Michael Stonebraker Michael L. Brodie. *Migrating Legacy Systems.* Morgan Kaufmann Publishers Inc., 1995.

[OH97] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA.* John Wiley & Sons, 1997.

[OHE97] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA.* John Wiley & Sons, 1997.

[Pro] Renaissance Project. Renaissance website, `http://www.comp.lancs.ac.uk/projects/renaissance/`.

[Sim] SimCenter Project Team. SimCenter website, `http://autofocus.informatik.tu-muenchen.de/stp97/`.

[SUN97a] SUN Microsystems. *The JDK 1.1.2 Documentation,* `http://java.sun.com/products/jdk/1.1/docs`, 1997.

[SUN97b] SUN Microsystems. *RMI – Remote Method Invocation,* `http://java.sun.com/products/jdk/1.1/docs/guide/rmi`, 1997.

[Tic85] Walter F. Tichy. RCS – A system for version control. *Software – Practice and Experience*, 15(7), July 1985.